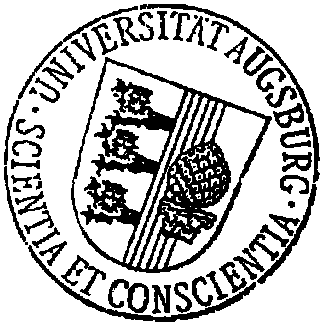
University of Augsburg
Diploma Thesis



Development of a PHP Compiler for Mono

Presented by:
Raphael Romeikat
Matriculation Number: 700543

Revisers:
Prof. Dr. Bernhard Bauer
Prof. Dr. Bernhard Möller
Institute of Computer Science
University of Augsburg

Deadline: February 6th, 2006

Contents

1 Preface

1.1 Incentives

Mono is an open source implementation of Microsoft's .NET platform including the tools, libraries and compilers required to build software on a variety of platforms: Linux, UNIX, MacOS X, Solaris and Windows. It is based on the Common Language Infrastructure (CLI), standardized by the ECMA standards group and ratified as standards by ISO. Similar to Java, the Mono environment consists of a compiler, virtual machine and API classes. The main compiler focus is on the C# language, but more and more languages are being supported by third party extensions. Mono is unique in many ways. It helps developers to use their existing knowledge on other platforms as libraries can be shared between languages. It brings together many modern programming language features. In particular the combination of them in a single environment makes Mono efficient to develop with.

PHP is a widely-used scripting language especially suited for web development and can easily be embedded into HTML. Its syntax and semantics are similar to other popular programming languages such as C, Perl and Java, therefore PHP can be used intuitively and easily. PHP also offers object oriented programming for better modularity and reuse of code. Another great strength is the availability of a very large number of library routines which makes coding of web pages very powerful. These libraries include a more or less complete set of system calls and access to a large variety of databases like Oracle and MySQL. A recent Netcraft survey indicates that PHP is used on almost 20 million domains worldwide, so it is the most widely used scripting language on the World Wide Web.

Given the popularity of Mono and PHP, why not take advantage of combining both? How could such benefits look like? With PHP's benefits on Mono many new possibilities would become reality. Mono developers could use PHP as a remarkably straightforward programming language outside of web pages. In this way they could also take advantage of the rapidly increasing range of useful PHP packages available.

1.2 Goals

We have now seen how popular and widespread Mono and PHP are. What I want to do is enable the Mono Community to take advantage of PHP's features. Therefore the main goal of this project is building a compiler that enables PHP scripts to run as standalone application on Mono. Before doing so major aspects of the compiler's design and implementation need to be specified, to the greatest extent possible at the very beginning.

1.2.1 Input and Output Language

As a compiler is basically a translator, it has an input and an output language. The question about input language can clearly be answered. Input to our compiler will be PHP scripts written in its current version 5.

Code being executable on Mono is written in Common Intermediate Language (CIL), so this will be the output language of our compiler. CIL is a low level human readable programming language and resembles an object orientated assembly language. It is entirely stack based and executed by Mono's virtual machine. Formerly it was known as Microsoft Intermediate Language (MSIL) and is now standardized by ECMA standards group and ratified as standards by ISO.

1.2.2 Functionality

The compiler should be able to translate all important built-in PHP language elements and functionalities. Extensions to PHP are widely used; however, they are not covered by this project. An exact specification about which PHP features are included and which ones are not can be found in chapter 7.1.

1.2.3 Process of Compilation

For the purpose of translating PHP to CIL we first need to develop a scanner separating the PHP source code into tokens. Secondly, a parser should build a syntax tree of the tokens already recognized by the scanner and should check syntactical correctness, to find out whether the code to be compiled is compliant with the PHP grammar. Thirdly, semantic checks on the code should be performed. The components described so far will be the front-end of the compiler.

If all the previous steps have been completed successfully the PHP source code should finally be translated to CIL to be executable on Mono. This translation will be the back-end of the compiler.

If any syntactic or semantic errors in the PHP source code occur during the process of compilation, they shall be reported in a user friendly and comprehensible way.

1.2.4 Implementation

The whole compiler should be written in C# as it is supposed to be compilable in Mono itself and as C# is presently the language best supported by Mono. If any third party software tools will be used to generate code, they need to generate C#. In addition, they must be under free license. The whole implementation of the compiler should be open source.

Concerning scanner and parser there are in general two ways of implementing them: hand coding or using a scanner and parser generator. The decision to be made should be based on aspects of convenience and efficiency. The same applies to the way of parsing, top-down or bottom-up. The final decision will be influenced by PHP and how its original parser is implemented. Another important aspect should be the ability to modify the compiler for future versions of PHP in the most practical way.

2 Existing Projects

There has already been some work undertaken for bringing .NET and PHP together. In this chapter I will give a short overview about projects that have been or are still worked on and compare them with regard to the focus of this thesis.

2.1 PHP Sharp

PHP Sharp [Know2004] is an open source PHP to CIL compiler by Alan Knowles partly implemented. To Alan, the motivation to build such a compiler seemed to make the best use of his unemployed time. He was primarily interested in dealing with the non typed language PHP mapping to the typed .NET bytecode system.

After a first investigation, Alan's first sense was that the compiler could be implemented quite easily in PHP. The challenge to build a self compiling PHP compiler motivated him additionally. He indeed succeeded in building the two core components, the scanner and the parser by modifying existing generator tools to output PHP code. However, as he delved deeper into the code generation part of the compiler, life became considerably more complex. This is why Alan changed his mind and started to work on a C# implementation. In doing so, he used a scanner generator tool called C# Lexer [Merr2002] outputting C# code. Soon after that he stopped working on the project. His last goal were to implement the `echo` command, variable and array support. However, as there is no code generating backend available his compiler cannot be used.

Nevertheless the compiler's source code is available and Alan describes some of his ideas about translation of PHP language elements in detail which in the end proved to be useful. Concerning the scanner and parser generator tools I will not follow Alan's strategy. Firstly, modifying existing tools seems a tricky task to me with unpredictable side effects in the end, unless one really has deep knowledge about the implementation of the generator tool to be modified. Secondly, there are tools available offering much more features than the scanner generator he used in his C# implementation.

2.2 IronPHP

IronPHP [Girs2005] is an open source PHP to .NET compiler, runtime and class library by Ross Girshick. Its goals are compatibility with PHP5, efficiency and access to .NET assemblies from PHP and it is supposed to run wherever Mono runs. In this way the IronPHP project focuses to achieve the same objectives as the Mono PHP Compiler described in this thesis.

Ross planned to implement the major part of the compiler in C#, but he also intended to explore other paths such as wrapping the existing PHP library which is written in C and marshalling data across the boundary to CIL byte-code. Ross also had in mind to re-implement a number of the commonly used PHP extensions and to add support for MySQL and other databases.

The latest version of IronPHP is 0.0.1 and is still experimental. This release covers a procedural subset of PHP's functionality; up to now no object orientated features are implemented at all. It contains a hand coded recursive descend scanner and parser. For the next version 0.0.2 Ross plans to feature a new scanner rewritten from scratch and a parser based on the Mono jay parser. There is a source code distribution but documentation is not available. The project was last modified in February 2005.

2.3 Phalanger

Phalanger [BenA2005] is a serious and huge project on which seven people have been working since 2002. Its aim is to create a module enabling execution of PHP scripts on Microsoft's .NET platform. This module is cooperating with the ASP.NET technology enabling it to generate web-pages written in PHP the same way ASP.NET pages are. Additionally, there is an integration module provided to develop PHP applications from within Microsoft's Visual Studio.NET which also offers practical features such as syntax highlighting.

In fact Phalanger compiles PHP5 scripts to MSIL. Built-in PHP functions are re-implemented in C# to enable full functionality of existing PHP scripts without any modification. Their implementation is located in the Phalanger Class Library which also includes the implementation of PHP's types such as **`PHP.PhpStrings`** and **`PHP.PhpArrays`**. External functions deployed in dynamic link libraries are loaded by an Extension Manager which simulates the PHP interpreter environment to the hosted libraries, so their functionality is the same as in PHP. The developers proclaim that PHP scripts translated with Phalanger would gain efficiency compared to execution with the original PHP interpreter.

Scripts compiled by Phalanger can only be executed on .NET platform 1.1 but not on Mono because some parts are mixed assemblies containing native code. This is not supported in Mono, so Phalanger itself as well as the compiled scripts only run in Windows with Microsoft's .NET Framework installed.

The Phalanger project proves that PHP compilation is feasible. It will be carried forward; unfortunately, it is not open source with only the code of the Phalanger Base Class Library being available publicly. Furthermore, one does not obtain any further internals or details about design and implementation, such as how scanner and parser work, whether generator tools are used or not. This is why it is very hard to compare Phalanger's architecture to the one developed in this thesis for the Mono PHP Compiler.

3 Mono

Some initial information about Mono has already been provided in the preface. I will now explain in more depth what Mono is, where it comes from and how its architecture looks like behind the scenes.

3.1 Introduction

Mono is an open-source implementation of the infrastructure upon which Microsoft's .NET Framework is built. It provides a compatible alternative including all technical features of .NET but avoiding the restrictive licensing and prohibitive costs that Microsoft imposes. You can indeed say that .NET is not just for Windows anymore. Apart from Windows, Mono provides the necessary infrastructure to develop and run .NET client and server applications on Linux, UNIX, MacOS X and Solaris. In this way Mono allows cross-platform programming and operating of .NET compatible applications. The main focus is on the C# language, but more and more languages are being and will be added. An advantage of Mono is the fact that it helps developers to use their existing knowledge on other platforms as libraries can be shared between languages. In particular, the combination of them in a single environment makes Mono efficient to develop with.

3.2 History

The Mono open-source project was initiated and co-financed by the Ximian Corporation for the development of an open source version of .NET Framework which is standardized by ECMA [ECMA2005]. Ximian was founded in 1997 as a provider of desktop and server solutions helping to enable enterprise Linux adoption all over the world. The effort quickly attracted a group of talented architects and engineers.

Some of their experts saw the presence of a UNIX compatible version as critical for the success of .NET. One of them was Miguel de Icaza, cofounder of Ximian. Internally at Ximian there was much discussion about building tools to increase productivity: making it possible to create more applications in less time and therefore reduce time and cost of development. In this respect Miguel became interested in the .NET technology as the moment the .NET documents came out in December 2000. At GUADEC 2001 Miguel showed a demo

for a few persons of their C# compiler and how it was able to parse itself. Motivated by positive results of a feasibility study Ximian finally decided to move staff away from other projects and constitute the Mono team. The Mono team did not have the ability to build a full .NET replacement on its own; thus, on July 19, 2001 the Mono open source project was officially announced at the O'Reilly conference.

Mono's initial purpose was to enable especially UNIX developers to build and deploy cross-platform .NET applications. Almost three years later, on June 30, 2004 Mono 1.0 was released. By the way Mono is the Spanish word for monkey; apart from the fact that Mono developers like monkeys there is no special reason for the choice of this project title [MonF2005].

Since then development has moved on rapidly. The current stable version of Mono is 1.1.12. It contains an implementation of Microsoft's .NET Framework 1.1 now being almost complete. Mono today consists of a C# compiler, a virtual machine and general, UNIX and Gnome specific class libraries. In addition, a complete Development IDE called "Mono Develop" is available. Support for platforms is no more limited to UNIX and Windows; implementations for Linux, MacOS X and Solaris have been added in the meantime.

Today Ximian is a part of and therefore sponsored by Novell. It has an active and enthusiastic contributing community and is positioned to become the leading choice for development of Linux and cross-platform applications.

3.3 Architecture

In this chapter I will discuss details about Mono's architecture. As indicated above Mono consists of

- a C# compiler
- a virtual machine with an advanced code generation engine for just-in-time compilation and pre-compilation of code
- class libraries to support ASP.NET WebForms, WebServices, ADO.NET databases and Windows.Forms
- UNIX and Gnome specific libraries

The architecture of the Mono environment and the way how the above mentioned parts interact is similar to the architecture of Microsoft's .NET Framework. The following diagram provides some guidelines where a given application fits into the Mono and .NET framework respectively. Layers one and two represent Mono itself; the other layers represent the environment Mono is embedded into.

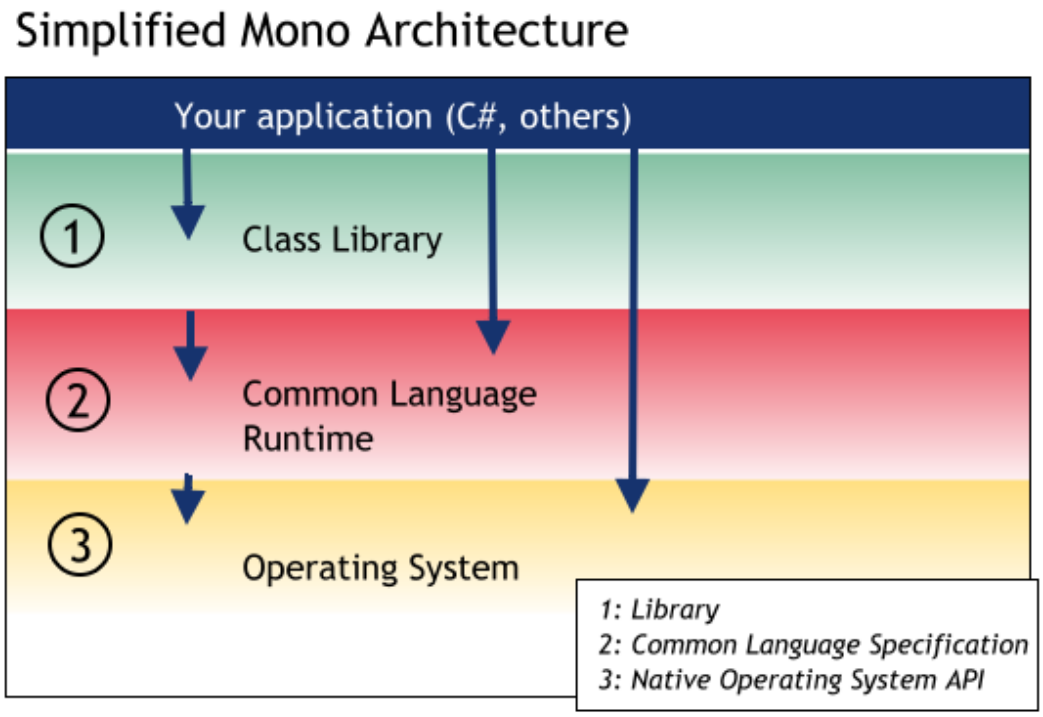


Figure 1: Simplified Mono Architecture [MonA2005]

3.3.1 Class Library

The class library provides a comprehensive set of facilities for application development. Primarily written in C#, it can be used by any language, due to the Common Language Specification. It is structured into Namespaces and deployed in shared libraries known as assemblies. When discussing the .NET framework, primarily reference is made to this class library.

Namespaces are a mechanism for logically grouping similar classes into a hierarchical structure. This prevents naming conflicts. The structure is implemented using dot-separated words. The top level namespace for most of the .NET framework is **`System`**. Under the **`System`** namespace you'll find namespaces such as **`System.IO`**, **`System.Reflection`**, **`System.Threading`**, **`System.Net`** and **`System.Net.Sockets`** [MonL2005]. There are other top-level namespaces as well, for example **`Accessibility`** and **`Windows`**. You can also create your own namespaces; a convention hereby is prefixing them with the name of your organization. **`Microsoft.VisualBasic`** is an example. This helps to prevent naming conflicts.

Assemblies are the physical packaging of the class libraries. These are **`.dll`** files, not to be confused with Win32 shared libraries. Examples are **`mscorlib.dll`**, **`System.dll`**, **`System.Data.dll`** and **`Accessibility.dll`** [MonL-2005].

3.3.2 Common Language Runtime

The Common Language Runtime (CLR) is the virtual machine that executes compiled .NET applications. It is a stack machine offering a special set of instructions written in Common Intermediate Language (CIL). CIL is an intermediate language that is abstracted from the platform hardware, so to say an analogue to Java Bytecode. As CIL is the target language of the Mono PHP compiler, we will have a close look on its syntax and semantics in chapter 5.

CLR and CIL together are an implementation of the Common Language Infrastructure (CLI) specification which is standardized by ECMA [ECMA2005]. This specification defines an environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures.

To be executable by the CLR an application must first be compiled to CIL as this is the language which is executed by the CLR. This is exactly what Mono's C# compiler does; it translates C# source code to CIL. When such code is executed, the platform-specific Virtual Execution System (VES) will compile the CIL to the machine-language according to the specific hardware.

Within a pure .NET/Mono application, all code is called managed. This means the application is governed by the CLI's style of memory management and thread safety. .NET/Mono applications can also use native legacy code, which is called unmanaged, by using the **`System.InterOpServices`** libraries to create C# bindings. Many of the libraries which ship with Mono actually use this feature of the CLI; in particular, the Gtk# libraries are C# wrappers around the underlying C libraries.

4 PHP

In this chapter I will give a comprehensive overview of the compiler's input language PHP. This includes general introduction, history, details about architecture and language reference. The latter one deals with PHP's syntax and semantics which are very important for correct translation to CIL in the end. In order to become familiar with PHP a couple of code examples are included as well.

4.1 Introduction

PHP is a powerful scripting language especially suited for developing server-side applications and dynamic web content. It allows its code to be embedded into HTML pages as you can see the following example.

```
<html>
  <body>
    <?php
      echo "Hello world!";
    ?>
  </body>
</html>
```

Example 1: Basic PHP Script

What distinguishes PHP from other scripting languages like JavaScript is the fact that code is executed on the server which makes data processing fast and efficient. Like that the client only receives the results of running that script with no way of determining what the underlying code may be. If a server processed the example above, output for a client would just be:

```
<html>
  <body>
    Hello world!
  </body>
</html>
```

Example 2: Output of a basic PHP Script

In addition, there is a vast number of external APIs available on the World Wide Web offering rich functionality, such as access to system calls, to databases, handling XML documents, communicating across applications via Microsoft's Component Object Model etc. Syntax and naming of language elements are very similar to the well-known languages Perl and C, which makes coding convenient for experienced programmers. Its current version comes with a brand new object model offering powerful object orientated features in the same way other popular languages like Java and C# do. A detailed language reference can be found further down in chapter 4.4.

4.2 History

This chapter about the history of PHP is mainly taken from [PhpH2005]. PHP succeeds an older product, named PHP/FI. PHP/FI was created by Rasmus Lerdorf in 1995, initially as a simple set of Perl scripts for tracking accesses to his online resume. He named this set of scripts "Personal Home Page Tools". As more functionality was required, Rasmus wrote a much larger C implementation which was able to communicate with databases and enabled users to develop simple dynamic web applications. Rasmus chose to release the source code for PHP/FI for everybody to see, so that anybody can use it, as well as fix bugs in it and improve the code.

PHP/FI, which stood for "Personal Home Page / Forms Interpreter", included some of the basic functionality of PHP as we know it today. It had Perl-like variables, automatic interpretation of form variables and HTML embedded syntax. The syntax itself was similar to that of Perl, albeit much more limited, simple, and somewhat inconsistent.

By 1997, PHP/FI 2.0, the second write-up of the C implementation, had a cult of several thousand users around the world, with approximately 50,000 domains reporting as having it installed, accounting for about 1% of the domains on the Internet [PhpH2005]. While there were several people contributing bits of code to this project, it was still at large a one-man project. PHP/FI 2.0 was officially released only in November 1997 after having spent most of its life in beta releases. It was succeeded shortly afterwards by the first alphas of PHP 3.0.

The whole new language was released under that new name which removed the implication of limited personal use that the PHP/FI 2.0 name held. The meaning of its new plain name PHP is a recursive acronym – PHP: Hypertext Preprocessor.

PHP 3.0 was the first version that closely resembles PHP as we know it today. It was created by Andi Gutmans and Zeev Suraski in 1997 as a complete re-write, after they found PHP/FI 2.0 severely underpowered for developing an eCommerce application they were working on for a university project. In an effort to cooperate and start building upon PHP/FI's existing user-base, Andi, and Zeev decided to cooperate and announce PHP 3.0 as the official successor of PHP/FI 2.0, and development of PHP/FI 2.0 was mostly halted.

PHP 3.0 was officially released in June 1998, after having spent about nine months in public testing. One of its biggest strengths was its strong extensibility features. In addition to providing end users with a solid infrastructure for lots of different databases, protocols and APIs, PHP 3.0's extensibility features attracted dozens of developers to join in and submit new extension modules. Arguably, this was the key to PHP 3.0's tremendous success. Other key features introduced in PHP 3.0 were the object oriented syntax support and the much more powerful and consistent language syntax. By the end of 1998, PHP grew to an install base of tens of thousands of users estimated and hundreds of thousands of web sites reporting it as installed. At its peak PHP 3.0 was installed on approximately 10% of the web servers on the Internet.

By the winter of 1998, Andi Gutmans and Zeev Suraski had begun working on a rewrite of PHP's core. The design goals were to improve performance of complex applications and to improve the modularity of PHP's code base. Such applications were made possible by PHP 3.0's new features and support for a wide variety of third party databases and APIs, but PHP 3.0 was not designed to handle such complex applications efficiently.

The new engine, dubbed "Zend Engine" met these design goals successfully, and was first introduced in mid 1999. The term "Zend" was comprised of their first names, Zeev and Andi. Based on this engine and coupled with a wide range of additional new features, PHP 4.0 was officially released in May 2000, almost two years after its predecessor PHP 3.0. In addition to the highly improved performance of this version, PHP 4.0 included other key features such as support for many more Web servers, HTTP sessions, output buffering, more secure ways of handling user input and several new language constructs.

PHP 5 was released in July 2004 after long development and several pre-releases. It is mainly driven by its core, the Zend Engine 2.0 with a new object model allowing for better performance and more features. In previous versions of PHP, objects were handled like primitive types such as instance integers and strings. The drawback of this method was that the whole object was copied when a variable was assigned or passed as a parameter to a method. In the new approach, objects are referenced by handle which is the object's identifier, and no more by value.

Today PHP is being used by hundreds of thousands of developers estimated and several million sites report as having it installed, which accounts for over 20% of the domains on the Internet. A survey carried out monthly by Netcraft shows PHP's popularity and how it is still increasing. In September 2005 PHP was used by web pages on 23,299,550 domains and 1,290,179 unique IP addresses [PhpU2005].

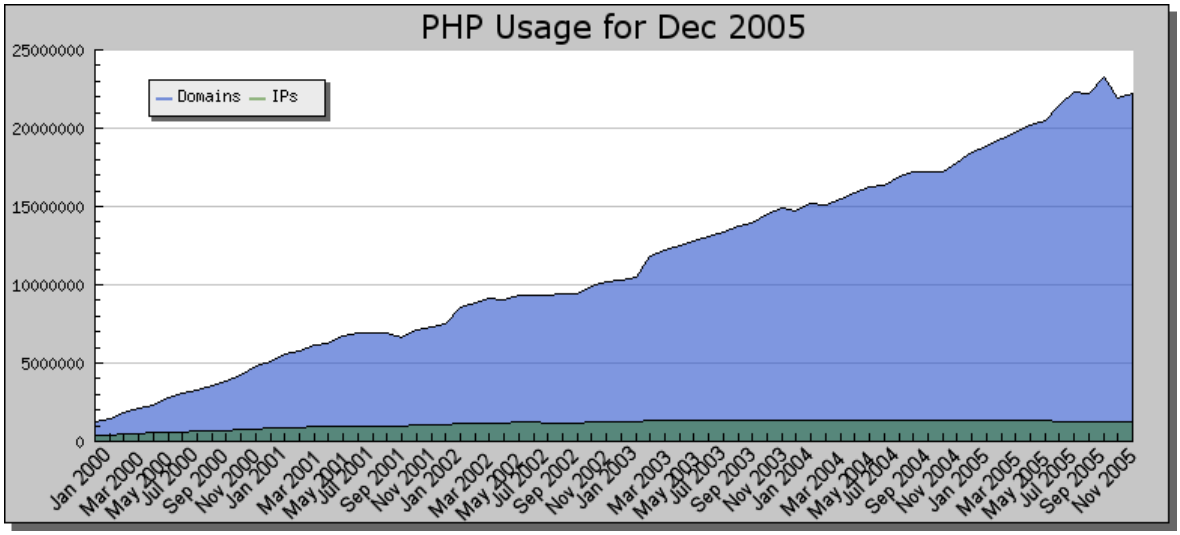


Figure 2: PHP Usage Statistics [PhpU2005]

4.3 Architecture

The Zend Engine is the basic scripting engine which drives PHP. Owned by Zend Technologies, the engine is licensed to PHP for free use under the Q Public license [Trol1999]; it is cross-platform and open source.

The diagram below shows the architecture and basic data flow for PHP-based web pages. The browser initiates a call to the web server, which passes the request through to the PHP web server interface (1). The web server interface calls the Zend Engine (2), which accesses the web server disk. Then the web server disk retrieves the code for the PHP based web page and sends the PHP code to the runtime compiler (3). Afterwards the latter creates a compiled representation of the script which is passed on to the Zend Engine Executor (4). The executor generates the HTML to be viewed by the browser. If there are any calls to the other modules such as SQL database calls, XML or Java, the Zend Engine passes them through the PHP modules, processes the requests and passes the results to the Web Server Interface (5), which finally sends the HTML to the browser (6) [ZenT2005].

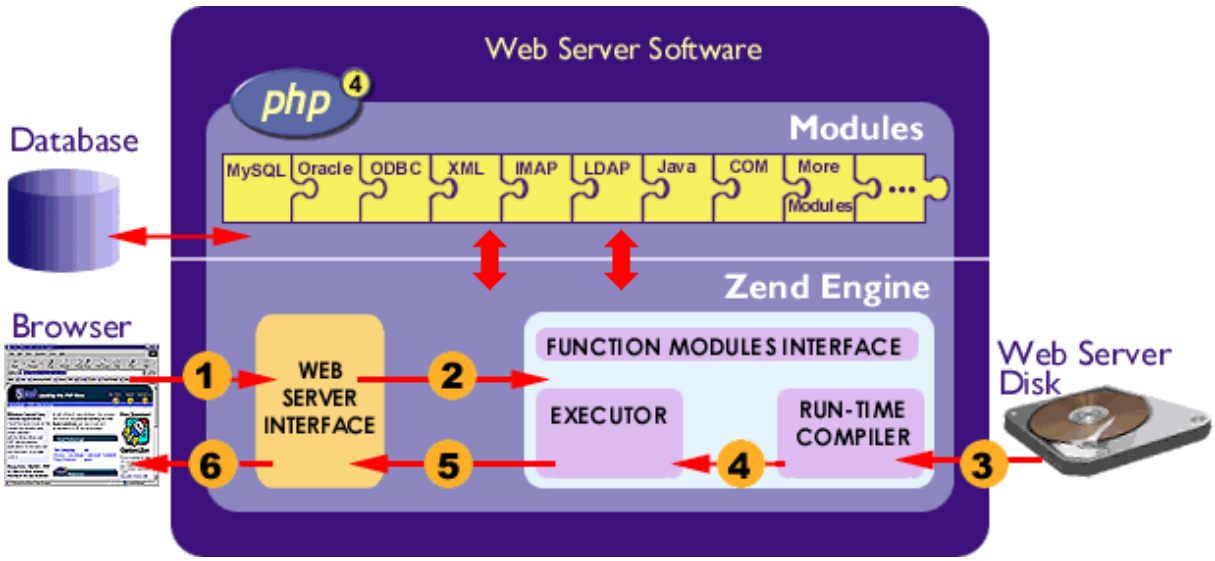


Figure 3: Web Server Software [ZenT2005]

As the PHP Compiler for Mono is intended to generate standalone applications, its architecture will be quite different from the Zend Engine, of course. I will discuss this matter in more detail in chapter 5.4.7.

4.4 Syntax and Semantics

As mentioned before PHP seems to be quite similar to other well-known languages. Unfortunately, no formal specification exists on PHP's syntax and semantics. The only official reference is Zend's PHP Manual [ZenM2005]. I'll now give an overview about syntax and semantics of PHP's language constructs including descriptive examples. This chapter cannot deal with all features PHP offers as this would go far beyond the scope of this thesis. The subset chosen contains the most important functionality, namely all features that will be translated to CIL later on. For more details about which functionality will be translated and which one will be skipped please refer to Appendix B. For a complete coverage please refer to [ZenM2005].

4.4.1 Statements

A PHP script consists of a series of instructions to be executed called statements. Statements are the minimum unit of structuring in imperative programming languages. A statement can be an assignment, a function call, a loop, a conditional statement of even an empty statement that does nothing. In addition, statements can be divided in statement groups by encapsulating a group of statements with curly braces. A statement group is a statement by

itself as well. In PHP, each non-grouped statement needs to be terminated with a semicolon at the end whereas the closing tag of a PHP code block automatically implies a semicolon.

```
<?php
  echo "This statement requires a semicolon";
  echo "This one doesn't"
?>
```

Example 3: PHP Statements

Statements contrast with expressions in that the former do not return results and are executed solely for their side effects, while the latter always return a result and often do not have side effects at all [WikS2005]. Expressions are discussed in more detail in chapter 4.4.6. All various statements types will be discussed in the following chapters.

4.4.2 Comments

PHP supports one-line and multi-line comments in different styles.

```
<?php
  echo "c1";  // This is a one line comment in C++ style
  echo "c2";  # This is a one line comment in shell style
  echo "c3";  /* This is a multi line comment
                 stretching across two lines */
  echo "This one doesn't"
?>
```

Example 4: PHP Comments

The one-line comment styles only comment to the end of the line or the current block of PHP code, whichever comes first. Multi line comments end by the first encountered **`*/`**. So be aware that simple nesting of such multi-line comments would not work.

4.4.3 Types

PHP supports four scalar types (**`boolean`**, **`integer`**, **`double`**, **`string`**), two compound types (**`array`**, **`object`**) and a special **`NULL`** type.

4.4.3.1 boolean

The easiest type is **boolean**. A boolean expresses a truth value, so it can be either **TRUE** or **FALSE**. These two keywords are also used to specify a boolean literal, both are case-insensitive.

4.4.3.2 integer

An **integer** is a number of the set Z = {..., -2, -1, 0, 1, 2, ...}. The size in bytes of an integer is platform-dependent; a maximum value of about two billion is the usual value which is 32 bits signed. PHP does not support unsigned integers. Integers can be specified in decimal (10-based), hexadecimal (16-based) or octal (8-based) notation, optionally preceded by a sign (– or +). Written formally as Lex Regular Expression [Levi1995] the possible structure for integer literals is:

```
decimal     : [1-9][0-9]*
            | 0
hexadecimal : 0[xX][0-9a-fA-F]+
octal       : 0[0-7]+
integer     : [+-]?{decimal}
            | [+-]?{hexadecimal}
            | [+-]?{octal}
```

Definition 1: PHP Integer

4.4.3.3 double

A **double** is a floating point number. Its size again is platform-dependent; a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value as this is 64 bit IEEE format [IEEE1985]. Doubles formally have the following syntax:

```
lnum          : [0-9]+
dnum          : ([0-9]*[\.]{lnum}) | ({lnum}[\.][0-9]*)
exponent_dnum : ({lnum} | {dnum}) [eE][+-]?{lnum}
```

Definition 2: PHP Double

4.4.3.4 string

A **`string`** is series of characters. In PHP, a character is the same as a byte; that is, there are exactly 256 different characters possible. This also implies that PHP has no native support of Unicode. A string can easily become very large, so there is no practical limit to the size of strings imposed by PHP. A string literal can be specified in three different ways: single quoted, double quoted and heredoc syntax.

The easiest way to specify a simple string is to enclose it in single quotes (`'`). To specify a literal single quote itself, it is necessary to escape it with a backslash. If a backslash is required before a single quote or at the end of the string, one will need to double it. Unlike the two other syntaxes, variables and escape sequences for special characters will not be expanded if they occur in single quoted strings.

If a string is enclosed in double-quotes (`"`), PHP will understand more escape sequences for special characters:

`\n`	linefeed (LF or 0x0A (10) in ASCII)
`\r`	carriage return (CR or 0x0D (13) in ASCII)
`\t`	horizontal tab (HT or 0x09 (9) in ASCII)
`\\`	backslash
`\$`	dollar sign
`\"`	double-quote
`\[0-7]{1,3}`	the sequence of characters matching the regular expression is a character in octal notation
`\x[0-9A-Fa-f]{1,2}`	the sequence of characters matching the regular expression is a character in hexadecimal notation

Table 1: PHP Escape Sequences

Another way to delimit strings is by using heredoc syntax (**<<<**). One should provide an identifier after the initial **<<<**, then the string and finally the same identifier to close the quotation. Here, the closing identifier needs to begin in the first column of the line. Heredoc text behaves just like a double-quoted string, without the double-quotes. This means that one does not need to escape quotes in your here docs, but you can still use the escape codes listed above. See some examples below to illustrate the three possibilities of specifying a string.

```php
<?php
  echo 'this is a simple string';
  $var = "string";
  echo "this is a $var with variable expansion";
  echo <<<EOD
    this is a string
    spanning multiple lines
    using heredoc syntax
  EOD;
?>
```

Example 5: PHP Strings

In this example a variable called **$var** is used. You may have noticed that a dollar sign precedes the variable name and its type is not specified explicitly. I will discuss PHP variables in chapter 4.4.4.

4.4.3.5 array

An **array** in PHP is actually an ordered map. A map is a type that maps values to keys. This type is optimized in several ways, so one can use it as a real array, or as a hashtable, as a list, stack, queue, collection and probably more.

An array can be created by the **array()** language construct. It takes a certain number of comma-separated **key => value** pairs. After creation one can access the array's values by using the common square bracket syntax enclosing a certain key.

```php
<?php
  $child1 = array(1 => "one", 2 => "two");
  echo $child1[2];  // "two"
  $child2 = array("three" => 3, "four" => 4);
  echo $child2["four"];  // 4
?>
```

Example 6: Accessing PHP Array Values

A key may be either an integer or a string. If a string key is the standard representation of an integer, it will be interpreted as such, so **"8"** will be interpreted as an integer **8**, while **"08"** will be interpreted as a string **"08"**. A value may be of any PHP type. So by having PHP arrays as values, one can also simulate trees.

```php
<?php
  $parent = array("child1" => $child1, "child2" => $child2);
  echo $parent["child1"][2];  // "two"
  echo $parent["child2"]["four"];  // 4
?>
```

Example 7: Simulating Trees in PHP

Without specification of a key for a given value, the maximum of the current integer indices will be taken; hereby the new key will be that maximum value plus one. By specifying a key already having a value assigned to it, that value will be overwritten.

```php
<?php
  // this array is the same as...
  array(1 => "one", "two", 3 => "nil", 3 => "three");
  // ...this array
  array(1 => "one", 2 => "two", 3 => "three");
?>
```

Example 8: PHP Array Keys

One can also modify an existing array by explicitly setting values in it or adding values to it. Both can be done by assigning values to the array while specifying the key in brackets. If the key already exists, the associated value will be overwritten, if the key does not exist, a new **`key => value`** pair will be added. You can also omit the key by using an empty pair of brackets (`[]`) which will again create a new key being the maximum of the existing integer keys plus one.

Each array holds an internal array pointer which always points to a certain array element. After creation of the array it is set to the first element of the array. One can access the current element pointed at by using the array function **`current`**. The array function **`next`** sets the pointer to the following element and returns it at the same time; **`prev`** does the same with the previous element respectively. **`reset`** sets the internal pointer back to the first element.

```php
<?php
  $arr = array(1 => "one", 2 => "nil");
  $arr[2] = "two";  // sets the value of key 2 to "two"
  $arr[3] = "three";  // adds the value "three" with key 3
  $arr[] = "four";  // is the same as $arr[4] = "four"
  echo current($array)  // returns "one"
  echo next($array)  // returns "two"
  reset($array)  // sets the pointer back to "one"
?>
```

Example 9: Internal PHP Array Pointer

4.4.3.6 object

This is the last but not least important data type. As more background knowledge about classes and PHP's object model is useful to deal with objects, I will describe the syntax and semantics about this data type later in chapter 4.4.10.

4.4.4 Variables

4.4.4.1 Basic Usage

Variables are a way of storing information for later use. In PHP, variables are represented by a dollar sign followed by the name of the variable. Variable names are case-sensitive and follow the same rules as other labels in PHP. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. In Lex Regular Expression Syntax [Levi1995] the specification is

```
variable_name : [a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*
```

Definition 3: PHP Variable Names

whereas **a-zA-Z** specifies a letter, **0-9** a digit, _ an underscore and **\x7f-\xff** all special ASCII signs from hexadecimal index 7F to FF. The syntax for assigning and using variables is very common and simple:

```php
<?php
  $var = 44;
  echo $var; // outputs 44
?>
```

Example 10: PHP Variable Usage

4.4.4.2 Reference Variables

With the syntax above variables are assigned by value. This means that if an expression is assigned to a variable, the entire value of the original expression will be copied into the destination variable. This also means that after assigning one variable's value to another variable, changing one of those variables will have no effect on the other.

However, PHP also offers assigning by reference resulting in the new variable becoming an alias for the original one. References in PHP are not like C pointers; they are symbol table aliases. Like that the same variable content can be accessed with different names. Changes to the new variable affect the original, and vice versa. No copying is performed and like that the assignment happens more quickly. To assign by reference, simply prepend an ampersand (`&`) to the beginning of the source variable which is being assigned as one can see in the following example.

```
<?php
  $a = 22;  // assign the value 22 to $a
  $b = &$a;  // reference $a via $b
  $b = 44;  // alter $b; $a is altered as well
  echo $a;  // outputs 44
?>
```

Example 11: PHP Reference Variables

Note that `$a` and `$b` are completely equal, not meaning `$a` pointing to `$b` or vice versa, but `$a` and `$b` pointing to the same place.

4.4.4.3 Variable Scope

The scope of a variable is the context in which it is defined. PHP variables specified outside a function have global scope and are available at each statement following. In order to make a variable unavailable from a certain point, an unsetting is required.

```
<?php
  $a = 1;
  // $a is available from now on
  unset($a);
  // $a is no more available from now on
?>
```

Example 12: Unsetting PHP Variables

Please note that unsetting a reference just breaks the binding between variable name and content. This does not destroy the variable content itself.

```
<?php
  $a = 1;
  $b = &$a;
  unset($a);  // won't unset $b, just $a
?>
```

Example 13: Unsetting PHP References

However, within user-defined functions (which are described in detail in chapter 4.4.9.1) a local function scope is introduced. Any variable defined inside a function is limited to the local function scope and variables defined outside that function are not available inside by default.

```
<?php
  $a = 1;  // global scope
  function foo() {
    $a = 2;  /* local function scope, so global $a
                is not overwritten by local $a */
  }
  echo $a;  // outputs 1
?>
```

Example 14: Local PHP Variable Scope

To change that default behavior the **`global`** keyword can be used. By declaring a variable global inside a function all references to it will refer to the global version.

```
<?php
  $a = 1;  // global scope
  function foo() {
    global $a;  // local $a now refers to global $a
    $a = 2;  // global $a is overwritten
  }
  echo $a;  // outputs 2
?>
```

Example 15: Global PHP Variable Scope

Another important feature of variable scoping is the **static** keyword. A static variable only exists in a local function scope, but it does not lose its value when program execution leaves this scope. If the scope is reentered, the variable will still have its old value. To declare a variable as static, just type the keyword in front of the variable name. In the following example every time **foo()** is called it will print the value of **$a** and increment it.

```
<?php
  function foo() {
    static $a = 0;
    echo $a;
    $a++;
  }
?>
```

Example 16: Static PHP Variables

Right here the **static** keyword is absolutely necessary. Without it this function would be useless as every time called it would set **$a** to **0** and print **0**. The **$a++** statement which increments the variable would serve no purpose as **$a** would disappear as soon as the function exits. In order not to lose track of the current count the variable **$a** must be declared as **static**.

4.4.4.4 Type Conversion and Casting

The type of a variable is not set by the programmer; rather it is decided at runtime by PHP depending on the context in which that variable is used.

```
<?php
  $bool = TRUE;  // a boolean
  $str = "foo";  // a string
  $int = 12;  // an integer
?>
```

Example 17: PHP Variable Type

An example of PHP's automatic type conversion is the addition operator (+). If any of the operands is a double, all operands will be evaluated as doubles and the result will be a double. Otherwise, the operands will be interpreted as integers and the result will be an integer as well. Note that this does not change the types of the operands themselves; the only change is in how the operands are evaluated.

```php
<?php
  $foo = "0";  // $foo is a string ("0")
  $foo += 2;  // $foo now is an integer (2)
  $foo += 1.3;  // $foo now is a double (3.3)
?>
```

Example 18: PHP Automatic Type Conversion

If one would like to force a variable to be converted to a certain type, one will need to cast it. Type casting in PHP works in the same way as in C: the name of the desired type is written in parentheses before the variable to be cast.

```php
<?php
  $int = 12;  // int is an integer
  $str = (string)$int;  // str is a string
?>
```

Example 19: PHP Casting

PHP basically allows casts and automatic type conversion from all and to all types making type juggling quite a possibility. A good example is string conversion to numbers. The string will evaluate as a double if it contains any of the characters `.`, `e`, or `E`. Otherwise, it will evaluate as an integer. At the same time the value is given by the initial portion of the string. If it starts with valid numeric data, this will be the value used. Otherwise, the value will be `0`. Valid numeric data is an optional sign, followed by one or more digits, optionally containing a decimal point and followed by an optional exponent. The exponent is an `e` or `E` again followed by one or more digits.

```php
<?php
  $foo = 1 + "10.5";  // $foo is a double (11.5)
  $foo = 1 + "-1.3e3";  // $foo is a double (-1299)
  $foo = 1 + "bob3";  // $foo is an integer (1)
  $foo = 1 + "bob-1.3e3";  // $foo is an integer (1)
  $foo = 1 + "10 USB Sticks";  // $foo is an integer (11)
  $foo = 1 + "10.0 Hard Drives";  // $foo is a double (11.0)
?>
```

Example 20: PHP Automatic Type Conversion

For conversion between all the other types please have a look into Zend's PHP Manual [ZenM2005].

4.4.5 Constants

A constant is an identifier for a simple value. As the name suggests, that value cannot change during execution. Constants are case sensitive and by convention, constant identifiers are always uppercase. The name of a constant follows the same rules as any label in PHP. A valid constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. Once a constant is defined, it can never be changed or undefined.

Constants are defined using the **`define()`** syntax specifying name and value as parameters. In doing so a **`boolean`** value will be returned informing whether the definition was successful. It will fail if a constant with the desired name has already been defined. Only scalar data (**`boolean`**, **`integer`**, **`double`** and **`string`**) can be used for a constant's value. To obtain the value of a constant, simply specify its name.

```
<?php
  define("PI", 3.14);
  echo PI;  // outputs 3.14
?>
```

Example 21: PHP Constants

See a summary about differences between constants and variables below:

- Constants do not have a dollar sign (**`$`**) before them
- Constants may only be defined using the **`define()`** syntax, not by simple assignment
- Constants may be defined and accessed anywhere without regard to variable scoping rules
- Constants may not be redefined or undefined once they have been set
- Constants may only evaluate to scalar values

4.4.6 Expressions

Expressions are the most important building stones in PHP. They are usually built from operands and operators and may be nested as well. In PHP, almost anything you write is an expression. The simplest yet most accurate way to define an expression is anything that has a value.

The most basic forms of expressions are constants and variables. When typing **`$a = 5`**, you're assigning **`5`** to **`$a`** using the assignment operator **`=`**. **`5`**, obviously, has the value **`5`**, or in other words **`5`** is an expression with the value of

`5`. After this assignment, one would expect `$a`'s value to be `5` as well, so if one wrote `$b = $a`, one would expect it to behave just as if you wrote `$b = 5`. In other words, `$a` is an expression with the value of `5` as well.

Slightly more complex examples for expressions are functions which are discussed in chapter 0. Functions are expressions with the value of their return value. Assume a function `foo()` always returning the value `5`. Then the statement `$c = foo()` behaves exactly in the same way as the statement `$c = 5`. In other words, `foo()` is an expression which evaluates to the value `5` in this case.

PHP takes expressions much further, in the same way many other languages do. Once more consider the example `$a = 5` we've already dealt with. It is easy to see that there are two values involved here, the integer value `5`, and the value of `$a` which is updated to `5` as well. But actually there is even one more value involved here, and that is the value of the assignment itself. The assignment itself evaluates to the assigned value, in this case `5`. In practice it means that `$a = 5`, regardless of what it does, is an expression with the value `5`. Thus, writing something like `$b = ($a = 5)` is like writing `$a = 5; $b = 5;`. Since assignments are parsed in a right to left order, you can also write `$b = $a = 5`.

Another good example is pre- and post-increment and -decrement which behaves like in many other languages. There are two types of increment, pre-increment and post-increment. Both essentially increment the variable with the effect on the variable being identical. The difference is with the value of the increment expression. Pre-increment, which is written `++$var` evaluates to the incremented value whereas post-increment, which is written `$var++`, evaluates to the original value of `$var` before it was incremented. With decrement the behavior is analogue, of course.

4.4.7 Operators

An operator is a kind of an action that is fed with one or more input values or expressions also called operands and which yields another output value. The construction of operator with operands itself is an expression again. There are three types of operators. Unary operators operate on only one value, binary operators take two operands and ternary operands operate on three operands. Furthermore, operators can be distinguished by their semantics. There are operators for arithmetic, assignment, bitwise operations, comparison, incrementing and decrementing, logical, string, array and type operations.

The precedence of an operator specifies how tightly it binds two expressions together. For example the value of the expression `1 + 2 * 3` is `7` and not `9` as the multiplication (`*`) operator has a higher precedence than the addition (`+`) operator. Parentheses may be used to force precedence. The following table lists the precedence of the PHP operators and is listed in descending order. As a result, an operator being higher up in the table means it has higher precedence than any latter one. Operators on the same line have equal precedence, in which case their associativity decides which order to evaluate them in.

Operators	**Associativity**	**Semantic Group**
`new`	non-associative	object/type
`[]`	right	array
`-> ::`	right	object/type
`++ --`	non-associative	in-/decrementing
`! ~ (bool) (int) (double) (string) (array) (object) instanceof`	non-associative	arithmetic, bitwise, logical, object/type
`* / %`	left	arithmetic
`+ - .`	left	arithmetic, string, array
`<< >>`	left	bitwise
`< <= > >=`	non-associative	comparison
`== != === !== <>`	non-associative	comparison, array
`&`	left	bitwise
`^`	left	bitwise
`|`	left	bitwise
`&&`	left	logical
`||`	left	logical
`?:`	left	comparison
`= += -= *= /= .= %= &= |= ^= <<= >>=`	right	assignment
`and`	left	logical
`xor`	left	logical
`or`	left	logical

Table 2: PHP Operators

I will now give an overview about all semantic groups of operators introducing each operator's name and semantics and including short examples in table form.

4.4.7.1 Arithmetic Operators

Arithmetic operators behave in the common way as you would expect from other programming languages as well.

Symbol	Name	Example	Result
`!`	Negation	`!$a`	Opposite of `$a`
`+`	Addition	`$a + $b`	Sum of `$a` and `$b`
`-`	Subtraction	`$a - $b`	Difference of `$a` and `$b`
`*`	Multiplication	`$a * $b`	Product of `$a` and `$b`
`/`	Division	`$a / $b`	Quotient of `$a` and `$b`
`%`	Modulus	`$a % $b`	Remainder of `$a` divided by `$b`

Table 3: PHP Arithmetic Operators

Please note that the division operator (`/`) returns a double value anytime, even if the two operands are integers. The remainder `$a % $b` is negative for negative `$a`.

4.4.7.2 Incrementing and Decrementing Operators

Incrementing and decrementing operators are available in pre and post version. As mentioned before both essentially increment the variable and decrement it respectively, but differ in the expression's result.

Symbol	Name	Example	Result
`++`	Pre-increment	`++$a`	Increments `$a` by one, then returns `$a`
`++`	Post-increment	`$a++`	Returns `$a`, then increments `$a` by one
`--`	Pre-decrement	`--$a`	Decrements `$a` by one, then returns `$a`
`--`	Post-decrement	`$a--`	Returns `$a`, then decrements `$a` by one

Table 4: PHP Incrementing and Decrementing Operators

4.4.7.3 Bitwise Operators

Bitwise operators allow you to turn specific bits within an integer on or off. If both the left- and right-hand parameters are strings, the bitwise operator will operate on the characters' ASCII values.

Symbol	Name	Example	Result
&	And	**$a & $b**	Bits that are set in both **$a** and **$b** are set
|	Or	**$a | $b**	Bits that are set in either **$a** or $b are set
^	Xor	**$a ^ $b**	Bits that are set in **$a** or **$b** but not both are set
~	Not	**~ $a**	Bits that are set in **$a** are not set and vice versa
<<	Shift left	**$a << $b**	Shift the bits of **$a $b** steps to the left (each step means multiplication by two)
>>	Shift right	**$a >> $b**	Shift the bits of $a $b steps to the right (each step means division by two)

Table 5: PHP Bitwise Operators

4.4.7.4 Logical Operators

Arithmetic operators just work in the common way, as you would expect.

Symbol	Name	Example	Result
and	And	**!$a**	**TRUE** if both **$a** and **$b** are **TRUE**
or	Or	**$a + $b**	**TRUE** if either **$a** or **$b** is **TRUE**
xor	Xor	**$a - $b**	**TRUE** if either **$a** or **$b** is **TRUE**, but not both
!	Not	**$a * $b**	**TRUE** if **$a** is not **TRUE**
&&	And	**$a / $b**	**TRUE** if both **$a** and **$b** are **TRUE**
||	Or	**$a % $b**	**TRUE** if either **$a** or **$b** is **TRUE**

Table 6: PHP Logical Operators

The reason for the two different variations of logical **and**/**&&** and **or**/**||** operators is that they operate at different precedences.

4.4.7.5 Comparison Operators

As their name implies comparison operators allow you to compare two values.

Symbol	Name	Example	Result
==	Equal	**$a == $b**	**TRUE** if **$a** is equal to **$b**

Symbol	Name	Example	Result
`===`	Identical	`$a === $b`	**TRUE** if `$a` is equal to `$b`, and they are of the same type
`!=`	Not equal	`$a != $b`	**TRUE** if `$a` is not equal to `$b`
`<>`	Not equal	`$a <> $b`	**TRUE** if `$a` is not equal to `$b`
`!==`	Not identical	`$a !== $b`	**TRUE** if `$a` is not equal to `$b`, or they are not of the same type
`<`	Less	`$a < $b`	**TRUE** if `$a` is strictly less than `$b`
`>`	Greater	`$a > $b`	**TRUE** if `$a` is strictly greater than `$b`
`<=`	Less or equal	`$a <= $b`	**TRUE** if `$a` is less than or equal to `$b`
`>=`	Greater or equal	`$a >= $b`	**TRUE** if `$a` is greater than or equal to `$b`
`?:`	Conditional	`$a ? $b : $c`	`$b`, if `$a` is **TRUE**; `$c`, otherwise

Table 7: PHP Comparison Operators

If you compare an integer with a string, the string will be converted to a numeric representation of its content. So if you compare two numerical strings, they will be compared as integers.

4.4.7.6 String Operators

There is only one string concatenation operator.

Symbol	Name	Example	Result
`.`	Concatenation	`$a . $b`	Concatenation of `$a` and `$b`

Table 8: PHP String Operators

4.4.7.7 Assignment Operators

The basic assignment operator is =. Notice that this is not a boolean equal to operator. In addition, there are combined operators for all of the binary arithmetic and string operators that allow you to use a value in an expression and then set its value to the result of that expression at the same time.

Symbol	Name	Example	Result
`=`	Assignment	`$a != $b`	`$a` gets set to `$b`'s value
`+=`	Addition	`$a -= $b`	`$a` gets set to `$a + $b`

Symbol	Name	Example	Result
-=	Subtraction	**$a *= $b**	**$a** gets set to **$a - $b**
***=**	Multiplication	**$a /= $b**	**$a** gets set to **$a * $b**
/=	Division	**$a %= $b**	**$a** gets set to **$a / $b**
%=	Modulus	**$a += $b**	**$a** gets set to **$a % $b**
.=	Concatenation	**$a .= $b**	**$a** gets set to **$a . $b**
&=	And	**$a &= $b**	**$a** gets set to **$a & $b**
|=	Or	**$a |= $b**	**$a** gets set to **$a | $b**
^=	Xor	**$a ^= $b**	**$a** gets set to **$a ^ $b**
<<=	Shift left	**$a <<= $b**	**$a** gets set to **$a <<= $b**
>>=	Shift right	**$a >>= $b**	**$a** gets set to **$a >>= $b**

Table 9: PHP Assignment Operators

The value of an assignment expression is the value assigned. That is, the value of **$a = $b** is **$b**'s value.

4.4.7.8 Object and Type Operators

PHP offers an operator for creating new objects of a specified type, one for accessing class members and methods and one for determining whether a given object is of a specified class.

Symbol	Name	Example	Result
new	New	**$a = new A()**	A new object of type **A**
->	Arrow	**$a->a**	**$a**'s member **a**
::	Scope Resolution	**$a::a**	**$a**'s static member **a**
instanceof	Instanceof	**$a instanceof A**	**TRUE**, if **$a** is of type **A** or of a subtype of **A**

Table 10: PHP Object and Type Operators

4.4.7.9 Array Operators

PHP offers a special set of operators for accessing array elements and unification and comparison of whole arrays.

Symbol	Name	Example	Result
[]	Access	**$a[$b]**	**$a**'s value for key **$b**
+	Union	**$a + $b**	Union of **$a** and **$b**

Symbol	Name	Example	Result
`==`	Equality	`$a == $b`	`TRUE` if `$a` and `$b` contain the same key/value pairs
`===`	Identity	`$a === $b`	`TRUE` if `$a` and `$b` contain the same key/value pairs in the same order and of the same types
`!=`	Inequality	`$a != $b`	`TRUE` if `$a` is not equal to `$b`
`<>`	Inequality	`$a <> $b`	`TRUE` if `$a` is not equal to `$b`
`!===`	Non-identity	`$a !=== $b`	`TRUE` if `$a` is not identical to `$b`

Table 11: PHP Array Operators

The union operator creates a new array and first appends the left handed, then the right handed array. While appending the right handed array duplicated keys are not overwritten but ignored.

For the array comparison operators elements of arrays are equal if they have the same key and the same value.

4.4.8 Control Structures

Control structures are one of the most important features of programming languages. They allow for conditional execution of code fragments, depending on the current state at runtime. `if` and `switch` statements are used to choose between different fragments of code to execute. The same fragment can be repeated multiple times with `for`, `foreach`, `while`, and `do` loops.

4.4.8.1 if

The `if` construct chooses a fragment of code to execute based on a certain condition. Its basic syntax is:

```
if (expression1)
  statement1;
elseif (expression2)
  statement2;
else
  statement3;
```

Definition 4: PHP if

At first, the expression following **if** is evaluated to its boolean value. If **expression** evaluates to **TRUE**, PHP will execute the subsequent inner statement; if it evaluates to **FALSE**, it will ignore it. There is only one **if** block allowed but multiple alternative **elseif** blocks and one optional default **else** block may follow. The **elseif** blocks will be processed in case the **if** expression does not match, one after another until an expression evaluates to **TRUE**, in which case the appropriate inner statement is executed. If neither the **if** nor one of the **elseif** conditions match, the **else** statement will be executed, if existent.

Often one would like to have more than one statement to be executed conditionally. Of course, there is no need to wrap each statement with an **if** clause. Instead, one can group several statements into a statement group using curly braces (**{ }**). Like that if statements can be nested indefinitely within other if statements as well.

Please also keep in mind that maximum one of the inner statements is executed because the subsequent **elseif** and **else** blocks will be ignored once an expression has matched. If no **else** block is available, possibly none of the inner statements will be executed.

```
<?php
  if ($a > $b)
    echo "a is bigger than b";
  elseif ($a == $b)
    echo "a is equal to b";
  else
    echo "a is smaller than b";
?>
```

Example 22: if

4.4.8.2 switch

The **switch** statement is similar to a series of **if** statements on the same expression. It is ideal in case one would like to compare the same variable or expression with a series of different values and execute a different piece of code depending on which value it equals to. Its basic syntax is:

```
switch (expression) {
  case expression1:
    statement1; break;
  default:
    statement2;
}
```

Definition 5: PHP switch

It is important to understand how the **switch** statement is executed in order to avoid mistakes. A **switch** construct executes statement by statement. In the beginning, no code is executed. As soon as a **case** statement is found with a value that matches the value of the **switch expression** PHP enters the statements immediately following. PHP continues executing the statements until the end of the whole **switch** block or the first **break** is reached. The special default case matches anything and therefore should succeed all case statements. It will be entered if no **break** occurs until then. Omitting a **break** at the end of a case statement will result in executing the statements of the following case. This may be intended though:

```
<?php
  switch ($i) {
    case 0:
    case 1:
    case 2:
      echo "$i is less than 3"; break;
    case 3:
      echo "$i is 3";
  }
?>
```

Example 23: switch

In a **switch** statement the condition is evaluated only once and the result is kept in mind and compared to each **case** statement. In an **elseif** statement series, the condition is evaluated again. This is why a **switch** statement is generally more efficient than an appropriate **if** statement being semantically identical.

4.4.8.3 while

The **while** loop is used to repeat a statement as long as a certain condition is met. This type of loop is the simplest one in PHP. Its basic syntax is:

```
while (expression)
  statement;
```

Definition 6: PHP while

The semantics of a **while** statement is simple. It executes the nested statement repeatedly, as long as the while **expression** evaluates to **TRUE**. The value of the expression is checked each time at the beginning of the loop, so even if this value changes during the execution of the nested statement, execution will not stop until the end of the current iteration. In case the expression evaluates to **FALSE** from the very beginning on, the nested statement is not even run once. As with other conditional statements, one can group multiple statements within the **while** loop by surrounding them with curly braces.

```
<?php
  $i = 0;
  while ($i < 10) {
    echo "loop still iterating";
    $i++;
  }
?>  // 10 iterations will be done
```

Example 24: PHP while

4.4.8.4 do

do loops are very similar to **while** loops offering the following syntax:

```
do
  statement;
while (expression)
```

Definition 7: PHP do

The only difference in semantics is the fact that the expression of a **do** loop is not checked in the beginning but at the end of each iteration. This is why the first iteration of a **do** loop is guaranteed to run, whereas the nested statement inside a **while** loop might not necessarily run at all.

```php
<?php
  $i = 0;
  do {
    echo "loop still iterating";
    $i++;
  }
  while ($i < 10)
?>  // 10 iterations will be done
```

Example 25: PHP do

4.4.8.5 for

The **for** loop is a more complex kind of statement in PHP. The syntax of a **for** loop is:

```
for (expression1; expression2; expression3)
  statement;
```

Definition 8: PHP for

for loops behave like their C counterparts. The first expression (**expression1**) is evaluated once and unconditionally at the beginning of the loop. In the beginning of each iteration, **expression2** is evaluated. If it evaluates to **TRUE**, the loop will continue and the nested statement will be executed. If it evaluates to **FALSE**, the execution of the loop will end. At the end of each iteration, **expression3** is evaluated.

```php
<?php
  for ($i = 0; $i < 10; $i++)
    echo "loop still iterating";
?>  // 10 iterations will be done
```

Example 26: PHP for

Each of the three expressions may be empty. **expression1** being empty simply means no code should be evaluated at the beginning; with **expression3** the same applies for the end of each iteration respectively. **expression2** being empty means the loop should run indefinitely. This feature may be useful in case you prefer ending the loop using a conditional **break** statement inside.

```php
<?php
  for ($i = 0; ; $i++) {
    if ($i < 10)
      echo "loop still iterating";
    else
      break;
  }
?>  // 10 iterations will be done
```

Example 27: PHP for expressions

4.4.8.6 foreach

PHP also offers a **foreach** construct, much like Perl and some other languages do. This feature gives an easy way to iterate over arrays. There are two syntaxes; the second is a minor but useful extension of the first one:

```
foreach (array_expression as $value)
  statement;

foreach (array_expression as $key => $value)
  statement;
```

Definition 9: PHP foreach

The first type iterates over the array given by **array_expression**. At the beginning of each iteration the value of the current array element is assigned to **$value** and the internal array pointer is advanced by one; so in the next iteration, the next element will be looked at. The second type behaves in the same way, except the fact that the current element's key will be assigned to the variable **$key** additionally.

When **foreach** first starts executing, the internal array pointer is reset to the first element of the array automatically, so you do not need to call **reset** before entering a **foreach** loop. By default **foreach** operates on a copy and not the array itself. Therefore changes to the array elements are not reflected in the original array. To operate on the original array elements instead of using copied ones you may reference the value variable by typing **&$value**. In any case the internal pointer of the original array is advanced with the processing of the array. Assuming the **foreach** loop runs to completion, the array's internal pointer will be at the end of the array.

```php
<?php
  $arr = array(1, 2, 3, 4);
  foreach ($arr as &$value)
    $value *= 2;
  // $arr is now array(2, 4, 6, 8)
?>
```

Example 28: PHP foreach

4.4.8.7 break/continue

As mentioned before the **break** statement ends execution of the current loop. It is available for the **switch**, **while**, **do**, **for** and **foreach** constructs.

The **continue** statement is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation followed by the next iteration. As with **break**, **continue** is available for the **switch**, **while**, **do**, **for** and **foreach** constructs.

4.4.8.8 Alternative Syntax for Control Structures

PHP also offers an alternative syntax for its control structures **if**, **switch**, **while**, **for** and **foreach**. In each case the alternative syntax is created by changing the opening brace of the inner statement block to a colon (:) and the closing brace to **endif;** and **endwhile;**, **endfor;**, **endforeach;**, **endswitch;** respectively.

```php
<?php
  $i = 0;
  while ($i < 10) :
    echo "loop still iterating";
    $i++;
  endwhile
?>  // 10 iterations will be done
```

Example 29: Alternative Syntax for PHP while

4.4.9 Functions

A function in programming is a sequence of code which performs a specific task, as part of a larger program. Functions can be called, thus allowing programs to access the function repeatedly without the function's code having been written more than once [WikF2005]. Functions may or may not return a value.

4.4.9.1 User Defined Functions

PHP allows user defined function definitions on top level of a script and within classes. The latter are preferably called methods and will be discussed in chapter 4.4.10. Top level functions are not located inside any class definition and thus have global scope; so they can be called from everywhere. A function may even be called before being defined.

It is also possible to call recursive functions in PHP. From theory there is no limit of recursion levels; however, in practice recursive calls with over 100 levels can smash the internal stack and cause a termination of the script. A function definition consists of the keyword **`function`**, a function name, a comma-separated list of function arguments and the function body being a sequence of statements encapsulated in curly braces:

```
<?php
  function recursion($a) {
    if ($a < 10) {
      echo "$a ";
      recursion($a + 1);  // recursive call
    }
  }
  recursion(5);  // outputs 5 6 7 8 9 10
?>
```

Example 30: User Defined PHP Functions

4.4.9.2 Function Arguments

Data may be passed to functions via the argument list, which is a comma-delimited list of expressions. PHP supports passing arguments by value, which is default behavior, passing by reference and default argument values.

4.4.9.3 Passing by Value/Reference

By default function arguments are passed by value, so if one changes the value of an argument inside a function, it will not change outside of the function. If one wishes to allow a function to modify its arguments, one should pass them by reference using an ampersand (**`&`**) in the function call. If one would like an argument of a function to always be passed by reference, one should prepend an ampersand to the argument name in the function definition.

```php
<?php
  function add_milk($tea) {
    $string .= " with milk";
  }
  function add_sugar(&$tea) {
    $string .= " with sugar";
  }
  $tea = "A cup of tea";
  add_milk($tea);
  echo $tea;  // outputs "A cup of tea"
  add_sugar($tea);
  echo $tea;  // outputs "A cup of tea with sugar"
?>
```

Example 31: PHP Passing Arguments by Reference

4.4.9.4 Default Arguments

A function may define default values for scalar arguments in C++-style. The default value must be a constant expression, so it must not be a variable, a class member or a function call, for example. For correct behavior default arguments of a function definition should precede all arguments not having a default value.

```php
<?php
  function make_tea($type = "black") {
    return "A cup of $type tea";
  }
  echo make_tea();  // outputs "A cup of black tea"
  echo make_tea("green");  // outputs "A cup of green tea"
?>
```

Example 32: PHP Default Arguments

4.4.9.5 Returning Values

A function may return a value or not. Values are returned by using the optional **`return`** statement within the function body. Any type may be returned, including arrays and objects. A **`return`** statement causes the function to end its execution immediately and pass control flow back to the line from which it was called. You cannot return multiple values from a function, but similar results can be obtained by returning an array containing the return values desired. To return a reference from a function instead of a value, an ampersand (**`&`**) must be used in both the function declaration and when assigning the returned value to a variable:

```php
<?php
  function &returns_reference() {
    return $some_reference;
  }
  $ref = &returns_reference();
?>
```

Example 33: PHP Returning Values

4.4.10 Classes and Objects

As mentioned before PHP 5 introduced a completely new object model with a completely rewritten way of object handling, allowing for better performance and more features than in previous versions. I will discuss all features of classes and objects PHP offers in the following sections.

4.4.10.1 General

In object-oriented programming, a class consists of a collection of types of encapsulated instance variables and types of methods, together with a constructor function that can be used to create objects of the class. A class describes the rules by which objects behave; these objects are referred to as instances of that class. A class specifies the structure of data of all instances as well as the methods which manipulate the data of the object; such methods are sometimes described as behavior. A method is a function with a special property having access to data stored in an object [WikC2005].

4.4.10.2 Class Declaration

In PHP, every class definition begins with an optional **`final`** declaration followed by the keyword **`class`**, a class name and the class body which is the definition of the class members and methods encapsulated in curly braces. A class can inherit members and methods of another class by using the **`extends`** keyword in the class declaration. It is not possible to extend multiple classes; a class can only inherit from one non-final base class. The structure of a typical class definition is demonstrated by the following example.

```php
<?php
  class Rectangle extends Quadrangle {
    // member declarations
    ...
    // method declarations
    ...
  }
?>
```

Example 34: PHP Class Declaration

4.4.10.3 Visibility

The visibility of a member or method is defined by prefixing the declaration with the keywords **public**, **protected** or **private**. Public items can be accessed from everywhere. Protected visibility limits access to the defining and inherited classes. Private visibility limits access only to the class that defines the item.

4.4.10.4 Member Declaration

The declaration of a class member consists of optional visibility, optional static declaration, class member name and an optional default value. Without visibility declaration the member will be treated as if it was declared as **public static**. There is no limit to the number of members a class may contain.

```php
    ...
    // member declarations
    public $length;
    public $width;
    ...
```

Example 35: PHP Class Member Declaration

Declaring a class member as **static** makes it callable from outside the object context. This means a static member cannot be accessed with a variable that is an instance of the object and cannot be re-defined in an extending class.

4.4.10.5 Constant Declaration

It is possible to define constant values on a per-class basis remaining invariable and unchangeable. Constants differ from regular class members in that the keyword **const** is used for declaration and the dollar symbol is not used to declare or use them. Constants are semantically identical to class members being declared as **public static**.

```php
<?php
  class Circle extends Shape {
    // member declarations
    public $radius;
    const PI = 3.14159265;
  }
?>
```

Example 36: PHP Class Constant Declaration

4.4.10.6 Method Declaration

Methods declarations consist of optional visibility, optional **`final`** declaration, method name and method body encapsulated in curly braces. You may skip the visibility when declaring a method; in this case the method is defined as **`public`**. The inherited methods and members can be overridden by re-declaring them within the same name, unless the parent class has defined a method as **`final`**.

```
    ...
    // method declaration
    public function area() {
      echo $this->length * $this->width;
    }
    ...
```

Example 37: PHP Method Declaration

Amongst all non-static methods a pseudo-variable **`$this`** is available. **`$this`** is a reference to the object context which the method belongs to. For static methods firstly **`$this`** is not available and secondly the same properties apply as for static members. A static method cannot be accessed from a variable pointing to an object instance and cannot be re-defined in an extending class.

If an extending class overrides the parent's definition of a method, PHP will not call the overridden method automatically. It is up to the extended class on whether or not the overridden method is called. If one would like it to be called, a call to **`parent::method_name()`** is required.

4.4.10.7 Constructor Declaration

A special method of a class is the constructor. It is invoked automatically whenever a new instance of the class is created, so it is suitable for any initialization the object may require before being used. A properly written constructor will leave the object in a valid state.

```
...
// constructor declaration
public function __construct($l, $w) {
  $this->length = $l;
  $this->width = $w;
}
...
```

Example 38: PHP Constructor Declaration

Parent constructors are not called implicitly. In order to invoke a parent constructor, a call to **parent::__construct()** within the child constructor is required.

4.4.10.8 Object Instantiation

To create a new instance of an object, the **new** keyword must be used and the instance created should be assigned to a variable:

```
<?php
  $rect = new Rectangle(2, 4);
?>
```

Example 39: PHP Object Instantiation

When assigning an already created instance of an object to a new variable, the new variable will access the same instance as the object that was assigned. A new instance of an already created object can be made by cloning which will be discussed in the next chapter.

4.4.10.9 Object Cloning

An object copy is created by using the **clone** keyword. When an object is cloned PHP will perform a shallow copy of all of the object's members. Any members being references to other variables will remain references. If a **__clone()** method is defined, the newly created object's **__clone()** method will be called to allow any necessary member modifications. An object's **__clone()** method cannot be called directly.

```
<?php
  $rect = new Rectangle(2, 4);
  $rect2 = clone $rect;
?>
```

Example 40: PHP Object Cloning

4.4.10.10 Accessing Class Members and Methods

There are basically two ways of accessing class members and methods: using the arrow operator (`->`) or the scope resolution operator (`::`).

The arrow operator is used for accessing a class member or method from an object context, which means from the point of view of a certain instance of a class, for example when an object has been created and assigned to a variable. In this case the selector operator must be used between the variable pointing to the object and the desired class member or method. When accessing class members in this way, the plain member name without the dollar sign (`$`) is used.

```
<?php
  $rect = new Rectangle(2, 4);
  echo $rect->length;  // outputs 2
  echo $rect->width;  // outputs 4
  echo $rect->area();  // outputs 8
?>
```

Example 41: Accessing PHP Class Members and Methods

As mentioned above **`$this`** is available within method bodies of non-static methods. It may be used with the arrow operator to access the class members and methods of the current class.

```
<?php
  class Rectangle extends Quadrangle {
    // member declarations
    public $length;
    public $width;
    // constructor declaration
    public function __construct($l, $w) {
      $this->length = $l;
      $this->width = $w;
    }
  }
?>
```

Example 42: PHP $this

The scope resolution operator is used for accessing static and constant members or methods of a class. In this case use the class name in front of the scope resolution operator. The special keyword **self** is available instead of the class name to access such items from inside the class definition. For access to overridden static and constant class members or methods of a parent class one may use the special keyword **parent** instead of the class name.

```
<?php
  class Circle {
    // member declarations
    public $radius;
    const PI = 3.14159265;
    // method declarations
    public function area() {
      echo self::PI * $this-> radius * $this-> radius;
    }
  }
?>
```

Example 43: PHP self

4.4.10.11 Comparing Objects

As one would expect from an object oriented language object comparison is not trivial in PHP. There are two operators for comparing objects, one for equality and one for identity.

On the one hand, when using the comparison operator (==), object variables are compared in a simple manner. This means that two object instances are equal if they have the same attributes and values, and are instances of the same class.

On the other hand, when using the identity operator (===), object variables are identical if they refer to the same instance of the same class. So, having two instances **o1** and **o2** of the same class, **o1 == o2** will return **TRUE**, **o1 === o2** will return **FALSE**. With two references to the same instance, both **o1 == o2** and **o1 === o2** will return **TRUE** whereas having two instances of two different classes both **o1 == o2** and **o1 === o2** will return **FALSE**.

5 CIL

Having discussed the compiler's input language the task is now to have a close look onto the compiler's output language CIL. This is what I will do in this chapter including a general introduction, history and language reference. The latter one talks about CIL's syntax and semantics which are again very important for correct translation from PHP. I'll once more include some illustrative examples in order to become more familiar with CIL.

5.1 Introduction

Common Intermediate Language represents a transient stage in the process of conversion of source code written in any .NET language to machine language. It is entirely stack-based and is an object oriented pseudo-assembly language standing between the source code and the machine code. It offers a CPU-independent set of instructions that can be efficiently converted to native code. Thus, languages which target Mono and the .NET Framework respectively must compile to CIL. During runtime CIL is then converted to CPU-specific code by the VES, usually using a just-in-time (JIT) compiler. As the CLR supplies JIT compilers for multiple platforms, the same set of CIL can be JIT-compiled and executed on any supported platform. A simple Hello world program would look like this in textual CIL:

```
.assembly extern mscorlib {}
.assembly HelloWorld {}
.method public static void Main() cil managed {
  .entrypoint
  .maxstack 1
  ldstr "Hello world!"
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Example 44: CIL Hello World

5.2 History

This chapter about the history of Common Intermediate Language is mainly taken from [Goug2002]. The idea of using an intermediate form within a pro-

gramming language compiler dates back at least to the 1970s. In an ideal situation the language dependent front-end would be entirely independent from the target hardware, while the code generating back-end would be sensibly independent from the particular language in which the source program was written. In his way the task of writing compilers for **n** languages on **m** machine architectures is factored into **n+m** compilers.

Many of these intermediate language representations were based on abstract stack machines. One particular representation, P-Code, was invented as an intermediate form for the ETH Pascal Compilers, but became pervasive as the machine code for the University of California San Diego (UCSD) Pascal System [UCSD1995]. What had been noted by the UCSD people was that a program encoded for an abstract stack machine may be used in two ways: a compiler back-end may compile the code down to the machine language of the actual target machine; or an interpreter may be written that emulates the abstract machine on the target. Usually the intermediate form uses only one byte per instruction, which is why such intermediate representations are often referred to as byte code forms. In the case of UCSD Pascal the code was so compact that the compilers could be run on the 4k of memory available on the very first microcomputers. As a consequence of this technology high-level languages became available for the first time.

The use of abstract machines as intermediate forms for conventional compilers has had its adherents. For example, previous Gardens Point compilers used a stack intermediate form called D-Code for all languages and platforms supported by the system. Although most implementations are fully compiled, a special lightweight interpreted version of the system was written in about 1990 for the Intel iapx86 architecture, allowing users with a humble IBM XT to produce the same results as the 32-bit UNIX platforms that the other implementations supported.

A largely failed attempt to leverage the portability of stack intermediate forms was the Open Software Foundation's Architecture Neutral Distribution Form (ANDF). The idea behind ANDF was to distribute programs in an intermediate form and complete the task of compilation during an installation step. The ANDF form was code for an abstract stack machine, but one with a slight twist. Generators of conventional intermediate forms, such as D-Code, have enough knowledge about the target's addressing constraints to be able to resolve object field accesses to address offsets, for example. In the case of ANDF the target is not yet known at the time of compilation, so that all such accesses must remain symbolic. In this way, instead of saying "the integer as the address top-of-stack plus 8" one must say "the field named foo of the object of class Bar whose reference is currently on the top of the stack." It has

been suggested that this incorporation of symbolic information into the distributed form was considered to be a threat to intellectual property rights by many software companies, which was a factor most contributing to the failure of the form to achieve widespread acceptance.

In the late 1990s Sun Microsystems released their Java language system. This system is, once again, based on an abstract stack machine. And again, like ANDF, it relies on the presence of symbolic information to allow such things as field offsets to be resolved at deployment time. In the case of Java and the Java Virtual Machine (JVM) the problem of symbolic content turned out to be a benefit. The presence of symbolic information is what allows deployment-time and runtime enforcement of the type system via the so-called bytecode verifier. These runtime type safety guarantees are the basis on which applet security is founded. In the meantime JVMs are available for all important platforms and Java tells a program portability story which transcends almost all other approaches.

In mid-2000 Microsoft revealed a new technology which became known as the .NET system. This technology consists of many components, but all of it depends on a runtime which is object orientated and fully garbage collected. As mentioned before, the runtime was named Common Language Runtime and processes an intermediate form called Microsoft Intermediate Language that is again based on an abstract stack machine.

Since standardization of C# and the Common Language Infrastructure in 2001 by ECMA [ECMA2005] and in 2002 by ISO, MSIL is officially known as Common Intermediate Language. Because of this legacy, however, CIL is still often referred to as MSIL, especially by longtime veterans of the .NET languages. Sometimes it is also referred to just as Intermediate Language (IL).

5.3 Architecture

The Common Intermediate Language consists of a base instruction set and an object model instruction set. The first one offers basic functionality for handling of data whereas the latter one deals with special object orientated features.

CIL instructions are executed by the VES which is an abstract stack machine and part of the CLR. It provides direct support for a set of built-in data types, a set of control flow constructs and an exception handling model. Let us have a closer look on data types and control flow constructs when discussing the instruction set in chapter 5.4.

The VES uses an evaluation stack which is theoretically unlimited in size. Elements can be pushed onto or popped from the stack. Pushing elements increases the size of the stack whereas popping elements decreases it. Other instructions may modify existing elements on the stack as well. There is a pointer called top which always points to the highest element and is only visible within the VES. One cannot modify top directly; the VES uses and modifies it implicitly when processing CIL instructions.

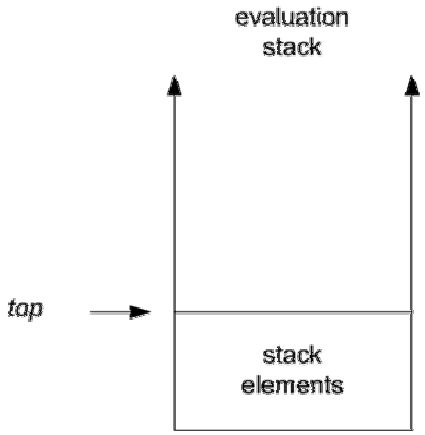


Figure 4: VES Evaluation Stack [ECMA2005]

When calling a method a so-called activation record is allocated containing the parameters passed to the method and the local variables visible within the method body. Both variables and methods are numbered separately and consecutively starting at zero. This is a purely logical numbering not depending on the size of the data types used. An activation stack may contain zero or more parameters and zero or more local variables.

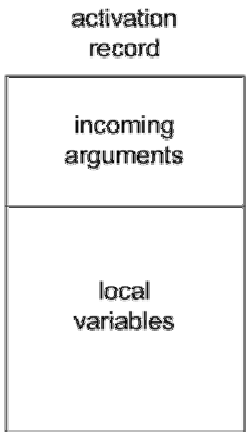


Figure 5: VES Activation Record [ECMA2005]

5.4 Syntax and Semantics

In this chapter I will give an overview about syntax and semantics of CIL's language constructs. This chapter cannot deal with all features CIL offers as this would go far beyond the scope of this thesis. However, all features that are used to perform the translation of PHP later on are discussed. For a complete coverage please refer to [ECMA2005].

An important term to describing a specific instruction is the stack delta (Δ). It describes the change in the number of elements on the stack. The instruction **`add`** for example has a stack delta of minus one, as it removes two elements from top of the stack and at the same time adds one new element:

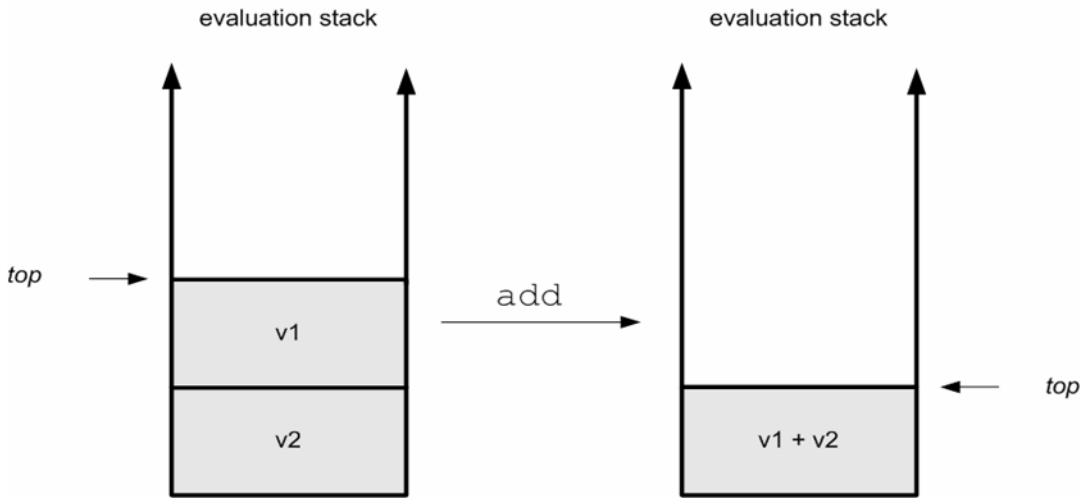


Figure 6: CIL add [ECMA2005]

A convenient and shorter notation of an instruction is the stack transition diagram. It shows the state of the evaluation stack before and after the instruction is executed. Below you can see the stack transition diagram for the **`add`** instruction.

`..., v1, v2 -> ..., res`

Definition 10: CIL Stack Transition Diagram

This diagram indicates that the stack must have at least two elements on it; in the definition the topmost value is called **`v2`** and the value underneath is called **`v1`**. In diagrams like this, the stack grows to the right, along the page. The instruction removes these values from the stack and replaces them by a result value, called **`res`**.

5.4.1 Types

The CLI supports a set of primitive data types given in the following table.

IL Name	Suffix	Description	Allowed on Stack
bool	**u1**	8-bit boolean	no
char	**u2**	16-bit Unicode character	no
int8	**i1**	8-bit signed integer	no
int16	**i2**	16-bit signed integer	no
int32	**i4**	32-bit signed integer	yes
int64	**i8**	64-bit signed integer	yes
native int	**i**	native signed integer	yes
unsigned int8	**u1**	8-bit unsigned integer	no
unsigned int16	**u2**	16-bit unsigned integer	no
unsigned int32	**u4**	32-bit unsigned integer	no
unsigned int64	**u8**	64-bit unsigned integer	no
native unsigned int	**u**	native unsigned integer	yes
float32	**r4**	IEEE 32-bit floating point	no
float64	**r8**	IEEE 64-bit floating point	no
F	**-**	native floating point	yes
O	**ref**	object type	yes

Table 12: CIL Types

The IL name gives the name of the type as it appears in textual CIL. The suffix gives the two-character suffix used to qualify instructions being specific to a certain type in the instruction set. For example, the instruction **conv.r4** converts the top element on the stack to the **float32** type, whereas **conv.u2** converts the top element to **char** or **uint16**. Instructions will be discussed adequately in the next chapter.

All data types given above can be manipulated using the CIL instructions. However, only a subset of these types is supported directly on the stack, as indicated in the last column of the table. If another type is pushed on the stack, it will be converted respectively. For example a **bool** value is actually an **int32** value on the stack, a bit pattern of all bits set to **0** representing **false** and one with at least one bit set to **1** representing **true**. If a numeric type shorter than 32 bytes is loaded onto the stack, it will be expanded by a set of usual unary conversions. Hereby integers shorter than 32 bits will be sign-extended to the **int32** type, while unsigned integers shorter than 32 bits will be zero-extended to **uint32**. The 16-bit wide character type **wchar** will also be

zero-extended to 32 bits, although the CLR will still know that it is a **wchar**. This behavior is similar to the language C, where all signed integers shorter than **int** are promoted to **int** before any operations are applied to them.

The treatment of **float32** and **float64** types also involves unary conversions. They are converted to the native float type **F** when being pushed on the stack. The native float type is whichever type is most efficiently supported by the underlying hardware, whereas this type must be precise enough to ensure that a round-trip conversion from and to **float64** is exact. On the Intel-x86 architecture, for example, the native float type would be the 80-bit temporary format used by the coprocessor.

Object references of type **O** are completely opaque. There are no arithmetic instructions that allow object references as operands, and the only comparison operations permitted are equality and inequality between two object references. There are no conversion operations defined on object references. Object references are created by the CIL object instructions **newobj** and **newarr**. They can be passed as arguments, stored as local variables, returned as values and stored in arrays and as fields of objects. Please also note that object references may be assigned the value **null** which is bit pattern of all bits set to zero on the stack.

Besides the types built-in directly in the CLR, the assembly **mscorlib** is available by default and contains some useful types derived from the general super type **System.Object**, such as **System.String** and **System.Array**. As their names say, the first one represents strings whereas the latter represents arrays. Object types are never pushed on the stack; instead a reference pointing to a specific object is pushed. Objects are always created on the heap.

5.4.2 Assemblies and Modules

Assemblies and modules are the grouping constructs in CIL. An assembly is the basic unit of deployment of a library or application; it is a set of one or more files deployed as a unit [DuBo2004]. An assembly always contains a manifest that specifies additional information such as version, culture, security information, which other files, if any, belong to the assembly, which other assemblies, if any, are linked to, etc.

A module is a single file containing executable content and usually part of an assembly. For an assembly to be executed rather than dynamically loaded as a library, an entry point is necessary in one of its modules defining where execution shall start.

Thus, the simplest kind of an assembly is one containing a single module. In textual CIL the syntax is:

```
.assembly HelloWorld {}
.module HelloWorld
```

Example 45: CIL Assembly and Module

If any metadata is present, it follows the assembly name and is enclosed in curly braces. Other assemblies may be linked from the current assembly in order to use their functionality. This can be done using the keyword **extern**. Often the built-in assembly **mscorlib** is referred to as it contains useful functionality such as the classes **System.Console** and **System.String**. The first one is used to output data onto the console; the latter one is used as built-in class representing strings as mentioned above.

```
.assembly extern mscorlib {}
.assembly HelloWorld {
  .hash algorithm 0x00008004  // security information
  .ver 0:1:0:0  // version of assembly
}
.module HelloWorld
```

Example 46: CIL Assembly Metadata

Please note that one-line comments in CIL can be added in C++ and Java style preceded by a double slash (//).

5.4.3 Classes

In CIL, classes are the grouping construct within a module. A class definition consists of a header declaration followed by the class body. The class body consists of a number of member declarations whereas a class member usually is a field or a method.

The class header starts with the keyword **.class** followed by a sequence of class attributes, an identifier that names the class and an optional class from which the current one inherits preceded by the keyword **extends**. A parent class referenced like that must be specified by its full name, also called dotted name, in case it is located in another module. The dotted name is the name of the enclosing assembly enclosed in square braces followed by all nested classes separated from each other by a dot and finally followed by the identifier of the class itself. A class neither declared as private nor as public will be

publicly visible by default. Class attributes which are possible are given in the following table:

Attribute	Effect
private	The class and its members are not visible outside of the enclosing assembly.
public	The class and its members are visible outside of the enclosing assembly.
abstract	The class cannot be instantiated.
interface	The class is an interface definition.
sealed	The class cannot be extended.

Table 13: CIL Class Modifiers

A typical class header in CIL might look as follows:

```
// class header
.class public Foo extends [mscorlib]System.Object {
  // class body empty
}
```

Example 47: CIL Class Declaration

5.4.4 Fields

Fields of a class contain data and may be either instance or static fields. Every object of a particular class has its own set of instance fields whereas static fields are shared between all instances of the class. If a field is not declared as static, it will be an instance field.

A field declaration starts with the keyword **.field** followed by a sequence of field attributes, the data type and the name of the field. Field attributes being possible are given in the following table:

Attribute	Effect
assembly	The field is only accessible within the enclosing assembly.
family	The field is only accessible in its class and in subclasses of its class.
private	The field is only accessible in its class.
public	The field is accessible publicly.
static	The field is static.
initonly	The field may only be assigned a value in constructors.

Table 14: CIL Field Modifiers

A typical field declaration might look as follows:

```
.class public Foo extends [mscorlib]System.Object {
  // field declaration
  .field public int32 i;
}
```

Example 48: CIL Field Declaration

In the instruction set there are different instructions for accessing values of static and instance fields. Static fields can be read using the **ldsfld** instruction and written using the **stsfld** one respectively. With instance fields the analogue instructions are called **ldfld** and **stfld** respectively. Both must be followed by the field's type, the field's class, a double colon and the field's name. You can see an example in chapter 5.4.5.4.

5.4.5 Methods

Methods contain executable code and are the callable units of CIL. All such callable units are called methods in CIL, whether they are static or not and whether they are value returning functions or proper procedures.

5.4.5.1 Method Header

A method declaration starts with the method header which consists of the keyword **.method** followed by a sequence of method attributes, the return type, the method name and a parenthesized list of the parameter's types. After that the method body follows enclosed in curly braces. There is a special return type called **void** for methods not returning an explicit value. The return type and the list of parameter types form the signature of the method. Method attributes which are possible are given in the following table:

Attribute	Effect
assembly	The method is only accessible within the enclosing assembly.
family	The method is only accessible in its class and in subclasses of its class.
private	The method is only accessible in its class.
public	The method is accessible publicly.
virtual	The method may be overwritten in subclasses.
final	The method cannot be overwritten in subclasses.
abstract	The method doesn't have an implementation.
instance	The method has a this pointer.

Attribute	Effect
static	The method is static.

Table 15: CIL Method Modifiers

A method may be exactly one of **static**, **instance** or **virtual**. Static methods do not have a **this** pointer. They are associated with a particular class, rather than an instance of a class. Non-virtual instance methods are associated with an object instance that becomes the **this** pointer of the call. If a method is declared to be **abstract**, then it must also be declared as **virtual**.

A typical method declaration might look as follows:

```
.class public Foo extends [mscorlib]System.Object {
  // method declaration
  .method public static void Main() {
    // method body empty
  }
}
```

Example 49: CIL Method Declaration

5.4.5.2 Constructors

Constructors are mainly used to initialize fields of the respective class. As static and non-static fields need to be treated differently constructors come in two flavors. There are instance constructors and static constructors. The instance one is called implicitly whenever a new instance of the respective class is created, the static one is called implicitly when the first instance is created. Constructors are defined exactly in the same way as methods are, with only the requirement for the name of an instance constructor to be **.ctor** and for a static one **.cctor**. Apart from that, a static constructor mustn't take any arguments. You can see an example in chapter 5.4.5.4.

5.4.5.3 Overloading and Overriding

Whenever a new method is introduced it is necessary to consider the effect of the new declaration on the accessibility of any other method named equally. There are two separate mechanisms at work: overriding and overloading.

In the object model virtual methods are inherited from the parent type of each class. If one dispatches a particular method on a particular receiver object and methods with same name and signature are available in several ancestor types, then the method of the receiver object's type will be invoked, or if there

is no such method, the matching method from the closest ancestor will be invoked. Whenever a new virtual method is declared it hides any equally named method with the same signature. This behavior is called overriding. The lookup algorithm described for invoking virtual methods is implemented using virtual method tables (v-tables) with each class. At class load time the entries in the v-tables are organized in a way that the effect of the lookup algorithm is obtained by a simple lookup in the v-table of the receiver object's class without the need to refer to the ancestors' v-tables. It is important to recognize that overriding occurs based on the name and signature of the method. If two methods are declared as having same name and parameter list, but differ in return value, they will occupy separate slots in the v-table. The term by the way for the example given is called return type covariance.

If two methods of a class have the same name but different signatures, the method's name is said to be overloaded. As described above, methods are matched on the basis of the complete name and signature. The possibility of having different methods for which the simple name is overloaded follows directly. By the way C# allows overloading but only on the number and type of the parameters in its current version. Overloading on the basis of the return type of the method is not allowed, although the CIL is able to support this.

5.4.5.4 Method Body

The method body contains the implementation of the method and is a sequence of method body items. Important items to appear are the **`.local`**, optional **`.entrypoint`** and **`.maxstack`** declarations. They may be placed anywhere within the method body. The **`.entrypoint`** declaration sets the entry point of the enclosing module to that line. **`.maxstack`** followed by an integer specifies the maximum height the stack may grow during execution of the method. Local variables used within the method body must be declared using the **`.locals`** item which is described in the next section.

In general, local variables are referred to by ordinal numbers, as are formal parameters. Formal Parameters have their numbers determined by their positions in the signature, numbering either from zero or from one. With instance methods, whether virtual or not, the receiver is always argument zero and the other parameters count from one. For static methods the parameters count from zero. Local variables are declared explicitly and numbered sequentially by default. A method may contain several local variable declaration items. The syntax starts with the keyword **`.locals`** followed by an optional **`init`** to initialize the locals declared with a default value and followed by a parenthesized list of desired types for the local variables to be declared.

A good example for a method using a local variable and accessing an argument and a field is the following constructor. It takes the argument passed and stores it to a local variable and to the field **a**.

```
.class public Foo extends [mscorlib]System.Object {
  .field private int32 a
  .method public void .ctor(int32) {
    .maxstack 2
    .locals init (int32)  // initialize a local variable
    ldarg.1               // load argument
    stloc.1               // store argument to local variable
    ldarg.0               // load receiver object
    ldarg.1               // load argument
    stfld int32 Foo::a    // store argument to field a
    ret                   // return
  }
}
```

Example 50: CIL Local Variables, Arguments and Fields

5.4.6 Base Instruction Set

The base instruction set contains basic operations. They are independent of the object model being executed and correspond closely to what would be found on a real CPU. I will divide the base instructions into semantic groups and discuss each group providing a concise example. The examples will consist of a small code fragment written in C# and the respective translation in CIL. A specification about all important instructions can be found in Appendix A.

5.4.6.1 Arithmetic Instructions

Arithmetic instructions are used to perform common calculations on numeric values. There are instructions for addition, subtraction, multiplication, division, negation and remainder. Say the following C# code fragment is given:

```
int a, b, c;
a = 2;
b = 4;
c = a + b;
```

Example 51: C# Arithmetic Operator

In CIL, the semantic equivalent is:

```
.locals init (int32, int32, int32)  // init local vars
ldc.i4.2  // load numeric constant 2
stloc.0   // store to local variable with index 0 (a)
ldc.i4.4  // load numeric constant 4
stloc.1   // store to local variable with index 1 (b)
ldloc.0   // load value of local variable with index 0 (a)
ldloc.1   // load value of local variable with index 0 (b)
add       // perform addition
stloc.2   // store to local variable with index 2 (c)
```

Example 52: CIL Arithmetic Instructions

5.4.6.2 Bitwise Instructions

Bitwise instructions are used to modify certain bits of a certain value. There are logical bitwise instructions and shift instructions. The latter ones can be used to efficiently multiply and divide by 2 which can be done in C# as follows:

```
int a = 4 << 1;
```

Example 53: C# Bitwise Operator

In CIL, the semantic equivalent is:

```
.locals init (int32)  // init local vars
ldc.i4.4  // load numeric constant 4
ldc.i4.1  // load numeric constant 1
shl       // shifts 4 left by 1 bit
stloc.0   // store to local variable with index 0 (a)
```

Example 54: CIL Bitwise Instructions

5.4.6.3 Control Flow Instructions

There are a couple of control flow instructions. They are used for jumping to a label specified in the current method depending on a certain test. A label is an identifier followed by a colon at the beginning of a line. Such labels are only allowed within method bodies. Control flow instructions are necessary for implementing conditional statements and loops of a higher language. Calling a static method and returning from a method are also considered to be a control flow instruction. Have a look onto the following example that, depending on the result of the `if`, returns `true` or `false`:

```
if (2 < 4)
  return true;
else
  return false;
```

Example 55: C# Control Flow Statements

In CIL, the semantic equivalent is:

```
ldc.i4.2         // load numeric constant 2
ldc.i4.4         // load numeric constant 4
bge else         // if 2 >= 4, jump to label else
ldc.i4.1         // load numeric constant 1 (for true)
br end           // jump to label end
else: ldc.i4.0   // load numeric constant 0 (for false)
end: ret         // return pushed value
```

Example 56: CIL Control Flow Instructions

When calling a static method it is necessary to use the dotted name preceded by the enclosed assembly. An example is calling the built-in static method for printing a string in C#:

```
System.Console.WriteLine("Hello world!");
```

Example 57: C# Static Method Call

In CIL, the semantic equivalent is:

```
ldstr "Hello world!"  // load string "Hello World!"
call void class [mscorlib]System.Console::WriteLine(string)
```

Example 58: CIL Static Method Call

5.4.6.4 Converting Instructions

Converting instructions are used to convert a value of a simple type to another simple type. Conversion from floating-point numbers to integral values truncates the number towards zero. When converting from a **`float64`** to a **`float32`**, precision may be lost. Let us have a look onto a simple **`float`** to **`int`** conversion in C#:

```
(int)2.4;
```

Example 59: C# Converting Operator

In CIL, the semantic equivalent is:

```
ldc.r8 2.4  // load numeric value 2.4
conv.i4     // convert 2.4 to int32
```

Example 60: CIL Converting Instruction

5.4.6.5 Load and Store Instructions

There are a couple of load and store instructions. The **ldc** ones are used for loading numeric constants with special and efficient versions for loading constants from **0** to **9**; loading arguments of the current method can be done using the **ldarg** instruction; the **ldloc** and **stloc** instructions are used for loading from and saving to local variables respectively. We have already seen several examples, most notably the one in chapter 5.4.5.4. There is a special **ldnull** instruction for pushing a null reference on the stack.

5.4.6.6 Logical Instructions

Logical instructions are used to compare to two values under a certain condition. If the check performed in the comparison is true, the value **1** will be pushed as **int32**; otherwise the value **0** will be pushed.

Say the following very short C# code fragment is given:

```
2 == 4;
```

Example 61: C# Logical Operator

In CIL, the semantic equivalent is:

```
ldc.i4.2  // load numeric constant 2
ldc.i4.4  // load numeric constant 4
ceq;      // check for equality
```

Example 62: CIL Logical Instructions

5.4.6.7 Other Instructions

There are several instructions that cannot be assigned to one of the previous semantic groups. These are a **dup** instruction duplicating the top element of the stack, a **pop** instruction doing just the opposite, namely removing the top element and last but not least a **nop** instruction doing nothing.

5.4.7 Object Model Instruction Set

The object model instructions are less built-in than the base instructions in the sense that they could be built out of the base instructions and calls to the underlying operating system. However, they provide a common and efficient implementation of object orientated services including field layout within an object and late bound method calls, for example.

5.4.7.1 Object Instructions

There are special instructions for creating an object of a specified type, for calling an object's virtual method, loading values from and storing them to a field and casting an object to a specified type. We've already seen how fields are accessed in chapter 5.4.5.4. The following C# example demonstrates how an instance of a class is created and a method of the newly created instance is called:

```
public class Foo {
  public static void Main() {
    new Foo().Bar();
  }
  public void Bar() {}
}
```

Example 63: C# Object Statement

In CIL, the semantic equivalent is:

```
.class public Foo extends [mscorlib]System.Object {
  // default constructor calling the parent's constructor
  .method public void .ctor() {
    .maxstack 1
    ldarg.0
    call instance void [mscorlib]System.Object::.ctor()
    ret
  }
  // method Main creating an instance of Foo and
  // calling the instance's method Bar()
  .method public static void Main() {
    .entrypoint
    .maxstack 1
    newobj instance void class Foo::.ctor()
    call instance void Foo::Bar()
    ret
  }
```

```
  // method Bar doing nothing
  .method public void Bar() {
    .maxstack 0
    ret
  }
}
```

Example 64: CIL Object Instructions

5.4.7.2 Array Instructions

CIL offers a special set of instructions supporting functionality concerning the built-in type **`System.Array`**. There are instructions for creating an array of a specified type, for loading from and storing to an element at a certain index and for calculating the length of an array. The following short C# code fragment includes such typical functionality:

```
int[] arr = new int[1];
int[0] = arr.length;
```

Example 65: C# Array Statements

In CIL, the semantic equivalent is:

```
.locals init (int32[])  // init local vars
ldc.i4.1      // load numeric constant 1 and
newarr int32  // create array of type int32 with that length
stloc.0       // store array to local variable
ldloc.0       // load array
ldc.i4.0      // load numeric constant 0
ldloc.0       // load array and
ldlen         // calculate length
stelem.i4     // store length to element at index 0
```

Example 66: CIL Array Instructions

6 Translation from PHP to CIL

As mentioned before the goal of this project is to build a compiler that enables PHP scripts to run on Mono. In order to do so, code written in PHP needs to be translated to CIL. In the previous two chapters I gave an overview about these two languages, now I will show how each of PHP's language constructs is translated to semantically equivalent CIL.

A large part of PHP's functionality is implemented in a runtime library which is actually implemented in C# and in a second step translated to CIL using the Mono C# Compiler. This is why any examples showing parts of the runtime functionality implementation are given in C# in this chapter. For more information on the runtime library and further details on the compiler's architecture please refer to chapter 7.

6.1 Statements and Comments

A PHP script consists of a series of statements. They are simply translated consecutively one after another. The translations for each individual statement type will be discussed in the following chapters.

PHP comments are neither adopted to CIL code nor translated at all but are just skipped by the compiler as they do not have any influence on execution of the script. Like that the size of the compiled application can be reduced by the amount of bytes that the comments occupy.

Assume the following abstract PHP script is given:

```
<?php
  // comment1
  statement1;
  // comment2
  statement2;
  // comment3
  statement3;
?>
```

Example 67: PHP Series of Statements and Comments

The respective instructions in textual CIL would be ordered in the same sequence skipping all comments:

```
<CIL instructions representing statement1>
<CIL instructions representing statement2>
<CIL instructions representing statement3>
```

Example 68: CIL Series of Instructions

6.2 Types

All PHP types are implemented as the CIL value and object types **System.Boolean**, **System.Int32**, **System.Double**, **System.String**, **PHP.Array**, **PHP.Object** and **null**. The super type **System.Object** is required as CIL is a strongly typed language whereas PHP is not. Wherever in a PHP source script data such as a variable or return type is used without any type specification, it will be typed **System.Object** in CIL.

Let us now have a look on the type **PHP.Array**. As arrays in PHP do not have a fixed length they cannot be implemented using CIL arrays. Instead variable length lists for keys and values are used. To append an element without specifying a key explicitly, the maximum integer key used so far must be kept in mind. The internal array pointer is implemented simply by using an integer pointing to the current element. When appending and removing elements both maximum key used and internal array pointer must be modified accordingly. However, outlining each method's implementation details would go far beyond the scope of this thesis. The complete source code is available at [Rome2005]. The basic implementation of the features mentioned is given as follows:

```
public class Array {
  public ArrayList keys;
  public ArrayList values;
  public int maxKey;
  public int current;
  public Array() : base() {
    keys = new ArrayList();
    values = new ArrayList();
    maxKey = -1;
    current = 0;
  }
  // append value
  public void Append(object value) {}
```

```
  // append value with key specified
  public void Append(object key, object value) {}
  // remove key
  public void Remove(object key) {}
  // get value for key specified
  public void Get(object key) {}
  // get key pointed to by internal array pointer
  public object Key() {}
  // get value pointed to by internal array pointer
  public object Current() {}
  // get value following Current()
  public object Next() {}
  // get value preceding Current()
  public object Prev() {}
  // get an array containing keys and values
  public object Each() {}
  // reset internal array pointer to 0
  public object Reset() {}
}
```

Definition 11: PHP.Array

Let us now have a look onto the object type **PHP.Object**. Each class defined individually in a PHP script will have **PHP.Object** as parent. An additional counter called **__id** is implemented whenever a new instance of an object type is created. This is needed for semantic equivalence to PHP's behavior of printing objects which will just output **Object** followed by the consecutive number of the instance. Thus, the basic implementation of the type **PHP.Object** is given as follows:

```
public class Object {
  public static int __maxId = 0;
  public int __id;
  public Object() : base() {
    __id = ++__maxId;
  }
  public override string ToString() {
    return "Object id #" + __id;
  }
}
```

Definition 12: PHP.Object

6.3 Variables

Handling variables the way PHP does is quite tricky. One issue is the fact one does not know at compile time whether a variable is defined or not. Say a variable **$var** is defined within an **if** block. In languages such as Java or C# **$var** would only be visible within that if block whereas in PHP it is still visible afterwards in case the **if** block was entered. In other words the existence of **$var** is dependent on evaluation of **<expr>** which cannot be determined at compile time.

```
<?php
  if (<expr>) {
    $var = 44;
  }
  // $var is still available
?>
```

Example 69: PHP Variable Scope

This is the reason why variables are managed at runtime in a virtual variable pool which is a container that tracks all variables and knows which piece of data each variable points to. All variables in the pool are grouped by their scope. There is a global scope for global variables and one scope for each instance and static function call encountered at runtime.

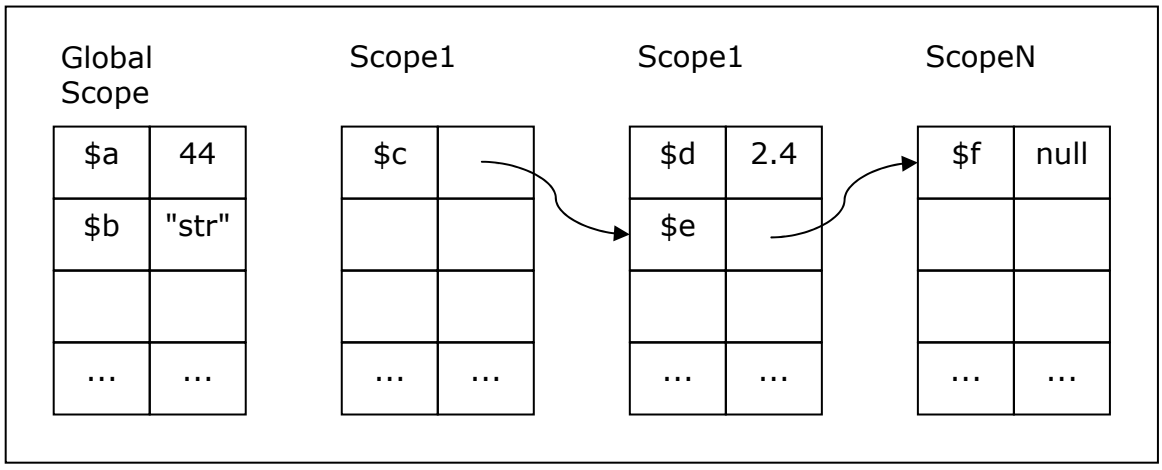


Figure 7: Variable Pool

As soon as a value is stored to a variable, an entry for that variable will be added to the variable pool if the variable has not been defined yet; otherwise the existing entry will be overwritten respectively. If a variable references another one a special data type **PHP.Reference** is used storing the place of the referenced variable. As soon as a variable is unset, its entry will be removed. All that is done by the static runtime method **void StoreToVariable(object value, object name)** which takes two arguments; the first one for the value to be stored, the second one for the name of the variable. There is another runtime method **object LoadFromVariable(object name)** which loads the value that has been stored to the variable name specified as argument. A variable can be unset using the runtime method **void UnsetVariable(object variable)**.

Assume the following PHP code is given:

```
<?php
  $var = 44;
  unset($var);
?>
```

Example 70: PHP Unsetting

The semantic equivalent in CIL is:

```
1:  ldc.i4.44
2:  box System.Int32
3:  dup
4:  ldstr "$var"
5:  call void class PHP.Runtime.Core::StoreToVariable
(object, object)
6:  pop
7:  ldstr "$var"
8:  call void class PHP.Runtime.Core::UnsetVariable(object)
```

Example 71: CIL Unsetting

In lines 1 and 2 a numeric constant **44** is loaded and boxed as value type **System.Int32**. The latter one is duplicated in line 3 as **44** is the result of the assignment expression **$var = 44;**. In line 4 a string **"$var"** is loaded. Now the two top elements on the stack are **"$var"** and **44**. Thus, the runtime method **void StoreToVariable(object, object)** can be called in line 5. In line 6 the result of the expression **$var = 44;** is removed again as it is not needed any more. In line 7 an object of type **string** representing **$var** is created again to unset the variable using the runtime method **void StoreToVariable(object, object)** in line 8.

Automatic type conversions are handled by the runtime methods implementing the PHP operators; they are discussed in chapter 6.5. If a variable is forced to be of a certain type using casting, an adequate static runtime method for converting the type will be called. Have a look on the following PHP code fragment:

```
<?php
  (int)2.4;
?>
```

Example 72: PHP Casting

The semantic equivalent in CIL is:

```
1:  ldc.r8 2.4
2:  box System.Double
3:  call int32 class PHP.Runtime.Convert::ToInt(object)
4:  pop
```

Example 73: CIL Casting

In this example a numeric constant 2.4 is loaded and boxed as value type **System.Double** in lines 1 and 2. In line 3 the adequate runtime converting method **int32 ToInt(object)** located in the namespace **PHP.Runtime.Convert** is called. Line 4 just pops the result of the expression from the stack as it is not required any more.

6.4 Constants

Handling of constants is quite similar to variables. However, constants are easier to manage as firstly they are always visible globally without regard scoping rules and secondly they cannot reference each other. Thus, there is just a simple constant pool holding name and associated value directly.

C1	2.4
C2	"abc"
...	...

Figure 8: Constant Pool

If a constant is defined an adequate entry in the constant pool is created using the static runtime method **`bool DefineConstant(object name, object value)`** located in the namespace **`PHP.Runtime.Core`**. Accessing a constant's name is done by using **`object GetConstant(object name)`**.

Like that the following PHP code

```
<?php
  define("PI", 3.14);
  echo PI;  // outputs 3.14
?>
```

Example 74: PHP Constant Definition

Is translated to CIL like that:

```
1:  ldstr "PI"
2:  ldc.r8 3.14
3:  box System.Double
4:  call class PHP.Boolean class PHP.Runtime.Core::
DefineConstant(object, object)
```

```
5:  pop
6:  ldstr "PI"
7:  call object class PHP.Runtime.Core::GetConstant(object)
10: call void class PHP.Runtime.Operators::Echo(object)
```

Example 75: CIL Constant Definition

In line 1 an object of type **`string`** representing **`"PI"`** as the constant's name is created. In lines 2 and 3 the same is performed with the constant's value **`3.14`** represented by an object of type **`System.Double`**. Now the runtime method **`bool DefineConstant(object, object)`** can be called in line 4. The result is not required and therefore popped in line 5. In line 6 once more an object of type **`string`** is created representing **`"PI"`** as the constant's name. Line 7 calls the runtime method **`object GetConstant(object)`** to retrieve the constant's value. Finally, in line 8 the runtime method **`void Echo(object)`** located in the namespace **`PHP.Runtime.Operators`** is called to output the result on the console.

6.5 Expressions and Operators

As mentioned earlier expressions are nested constructs and consist of operators and operands. The simplest kind of operands is numeric values, string values, variables or constants. They are translated as described in the previous chapters. In case operands are expressions as well, they are translated recursively starting at the most inner one, putting its result on the stack in order to use its value as operand for the enclosing expression and so on.

We have already seen how the assignment operator = is implemented; I will now discuss the remaining operators. As there is a great amount of operators in PHP, I cannot deal with each one separately; however, this is not necessary as the principle of operator translation is continuous and will become clear by means of an illustration. All remaining operators are implemented in C# as part of the runtime library, located in the namespace **`PHP.Runtime.Operators`**. Let us have a look on the implementation of the arithmetic operator **`*`**:

```
public static object Times(object o1, object m2) {
  Core.DeReference(ref o1, ref o2);
  if (o1 is Array || o2 is Array)
    throw Report.Exception(500);
  double d1 = Convert.ToDouble(o1);
  double d2 = Convert.ToDouble(o2);
  double result = f1 * f2;
  if (result % 1 == 0)
    return (int)result;
  else
    return result;
}
```

Definition 13: object Times(object, object)

First of all **`Times`** takes the two arguments **`o1`** and **`o2`** which are to be multiplied. As the result may be of type **`int`** or **`double`**, the super type **`object`** is used as return type. If one of the arguments or both are of type **`PHP.Array`**, an appropriate exception will be thrown as arrays are not allowed as operands for multiplication in PHP. Thereupon both arguments are treated as values of type **`duble`** to avoid separate handling of integer and floating point multiplication. Both arguments are multiplied and in dependence on the type of the result it is returned as **`int`** or **`double`**.

Say the following short PHP code fragment is given:

```
<?php
  2.2 * 5;
?>
```

Example 76: PHP Times

In CIL, the semantic equivalent is:

```
1:  ldc.r8 2.2
2:  box System.Double
3:  ldc.i4 5
4:  box System.Int32
5:  call object class PHP.Runtime.Operators::Times(object, object)
6:  pop
```

Example 77: CIL Times

In lines 1 and 2 an object of type **System.Double** representing **2.2** is created. In lines 3 and 4 the same is done with the value **5** represented by an object of type **System.Int32**. Now the runtime method **object Times(object, object)** is called in line 5.

Almost all other operands are implemented and translated respectively using appropriate and semantically identical C# functionality, regardless of arithmetic, bitwise, logical, comparison, string or array operators. However, the object operators **new**, **->** and **::** are translated in a different way which will be discussed in detail in chapter 5.4.7.

6.6 Control Structures

The main focus of this chapter is the translation of the PHP control structures. All of them require labels and branching when being implemented in CIL. There are two kinds of such statements: the conditional statements **if** and **switch** and the loop statements **while**, **do**, **for** and **foreach**.

6.6.1 Conditional Statements

Translating a conditional PHP statement isn't difficult in case branching is used effectively. Basically the two conditional statements **if** and **switch** are very similar. However, I'll just present how both of them are translated.

6.6.1.1 if

The following abstract code fragment contains all possible parts of an **if** statement.

```
if (expression1)
  statement1;
elseif (expression2)
  statement2;
else
  statement3;
```

Example 78: PHP if

In CIL, the semantic equivalent is:

```
1:  <CIL for evaluating expression1>
2:  call bool class PHP.Runtime.Convert::ToBool(object)
3:  brfalse 6
4:  <CIL for evaluating statement1>
5:  br 12
6:  <CIL for evaluating expression2>
7:  call bool class PHP.Runtime.Convert::ToBool(object)
8:  brfalse 11
9:  <CIL for processing statement2>
10: br 12
11: <CIL for processing statement3>
12:
```

Example 79: CIL if

First of all, **expression1** is evaluated pushing its result on the stack which can be seen in line 1. Now an instance of **object** is top of stack, but for the following branching instruction a CIL **bool** type is needed. This is why an adequate converting method is called in line 2. Right now the boolean result is checked in line 3. If **expression1** is **true**, the CLR will ignore the branching instruction and proceed with the next line of CIL code. In this case **statement1** is processed in line 4 and line 5 jumps unconditionally to the end of the whole if statement. If **expression1** is **false**, the branching instruction will transfer flow control to line 6 which is the beginning of the **elseif** part. The latter one is translated in the same way as the if part is with evaluating **expression2** in line 6, converting the result to a CIL **bool** value in line 7 and in line 8 jumping to the right line. If **expression2** is evaluated to **true**, the CLR will process **statement2** and jump to the end which can be seen in lines 9 and 10; otherwise a jump to the **else** part located in line 11 will be performed.

6.6.1.2 switch

See below an abstract example of a typical **switch** statement.

```
switch (expression) {
  case expression1:
    statement1; break;
  default:
    statement2;
}
```

Example 80: PHP switch

In CIL, the semantic equivalent is:

```
1:  <CIL for evaluating expression>
2:  dup
3:  <CIL for evaluating expression1>
4:  call class object class PHP.Runtime.Operators::IsEqual
(object, object)
5:  call bool class PHP.Runtime.Convert::ToValueBool(object)
6:  brfalse 9
7:  <CIL for processing statement1>
8:  br 11
9:  pop
10: <CIL for processing statement3>
11:
```

Example 81: CIL switch

First of all, **expression** is evaluated pushing its result on the stack which you can see in line 1. Lines 2 to 6 process the first case block; they evaluate **expression1**, check whether its result matches **expression** using adequate runtime methods and transfer control flow to the respective line. If the two expressions match, lines 7 and 8 will process **statement1** and jump unconditionally to the end; if not, a jump to the next block will be performed, which is the **default** block starting in line 9 in this case. The latter one pops the top of stack element which is still the result of evaluating expression and is not needed any more. Finally line 10 processes **statement2**.

6.6.2 Loop Statements

There are four kinds of PHP loop statements: **while**, **do**, **for** and **foreach**. I will discuss each one's translation by giving a typical abstract example and its semantic equivalent again.

6.6.2.1 while

The following code fragment represents the abstract example of a typical **while** statement.

```
while (expression)
  statement;
```

Example 82: PHP while

In CIL, the semantic equivalent is:

```
1:  <CIL for evaluating expression>
2:  call bool class PHP.Runtime.Convert::ToBool(object)
3:  brfalse 6
4:  <CIL for processing statement>
5:  br 1
6:
```

Example 83: CIL while

Translation of while is very straight forward. At first **expression** is evaluated in line 1 pushing its result on the stack. Lines 2 and 3 convert the result to a CIL **bool** value and depending on the result perform an adequate jump. If **expression** is **true**, lines 4 and 5 will process **statement** and jump back to the beginning; if **expression** is **false**, the loop will be exited by jumping to the end.

6.6.2.2 do

Translating a **do** loop is quite similar to a **while** one; the only difference is the fact that **do** ensures execution of the loop body for at least one time whereas **while** does not. A typical **do** loop looks like the following abstract code fragment.

```
do
  statement;
while (expression)
```

Example 84: PHP do

In CIL, the semantic equivalent is:

```
1:  <CIL for processing statement>
2:  <CIL for evaluating expression>
3:  call bool class PHP.Runtime.Convert::ToBool(object)
4:  brtrue 1
```

Example 85: CIL do

First of all, **statement** is processed in line 1. Lines 2 and 3 evaluate **expression** and convert it to a CIL **bool** value again. If **expression** is **true**, line 4 will jump back to the beginning; if **expression** is **false**, the loop will be exited.

6.6.2.3 for

For loops are more complex; however, translating is not very difficult. See the following code fragment representing the abstract form of a **for** statement.

```
for (expression1; expression2; expression3)
  statement;
```

Example 86: PHP for

In CIL, the semantic equivalent is:

```
1:  <CIL for evaluating expression1>
2:  <CIL for evaluating expression2>
3:  call bool class PHP.Runtime.Convert::ToBool(object)
4:  brfalse 8
5:  <CIL for processing statement>
6:  <CIL for evaluating expression3>
7:  br 2
8:
```

Example 87: CIL for

At first **expression1** and **expression2** are evaluated in lines 1 and 2 pushing their results on the stack. Afterwards the result of **expression2** is converted to a CIL **bool** value and depending on the result an adequate jump is performed in lines 3 and 4. If **expression2** is **true**, lines 5 to 7 will process **statement**, evaluate **expression3** and let control flow fall back to the beginning of the loop in line 2; if **expression** is **false**, the loop will be exited by jumping to the end.

6.6.2.4 foreach

Translating a **foreach** loop is rather laborious as handling of arrays is involved heavily, which implies calls to runtime array functionality in many places. The following code fragment represents the abstract example of a typical **foreach** statement.

```
foreach (array_expression as $value)
  statement;
```

Example 88: PHP foreach

In CIL, the semantic equivalent is:

```
1:  <CIL for evaluating array_expression>
2:  dup
3:  isinst PHP.Array
4:  brfalse 32
5:  dup
6:  callvirt instance class object class PHP.Array::Reset()
7:  pop
8:  ldnull
9:  ldstr "$value"
10: call void class PHP.Runtime.Core::StoreToVariable(class ob-
ject, object)
11: dup
12: callvirt instance bool class PHP.Array::CurrentIsValid()
13: brfalse 32
14: dup
15: callvirt instance object class PHP.Array::Current()
16: ldstr "$value"
17: call void class PHP.Runtime.Core::StoreToVariable
(object, object)
18: <processing statement>
```

```
19: dup
20: dup
21: callvirt instance object class PHP.Array::Key()
22: ldstr "$value"
23: call object class PHP.Runtime.Core::LoadFromVariable(object)
24: callvirt instance void class PHP.Array::Append(object, object)
25: dup
26: callvirt instance object class PHP.Array::Next()
27: pop
28: br 12
29: pop
```

Example 89: CIL foreach

At first **array_expression** is evaluated in line 1 pushing its results on the stack. Lines 2 to 4 ensure that the object being processed is of type **PHP.Array**; otherwise the whole loop is exited. Lines 5 to 7 reset the internal array pointer to the first element. In lines 8 to 10 a local variable named **$value** is created with no initial value assigned to it. After completing these initial actions the actual loop starts to process. Lines 11 to 13 check whether there are still array elements to be processed. If not, the loop will be exited by jumping to the end. If there are still array elements left, the value of the current array element will be stored to **$value** in lines 14 to 17 and can be used in processing **statement** in line 18. Lines 19 to 24 store the contents of **$value** back to the array at the position of the current element. Finally, in lines 25 to 28 the internal array pointer is advanced by one step and a jump back to the beginning of the actual loop to process the next element.

6.7 Functions

Functions are supported as methods by CIL. PHP top level functions that do not reside in a class are encapsulated in an internal class called **__MAIN** during compilation. For more information about translating classes please refer to chapter 5.4.7. Statements not located in any function but on top level of a PHP script are encapsulated in an internal method **__MAIN** which is the entry point of the whole translated application and is located in the class **__MAIN**.

Have a look on the following example showing how simple a method can be.

```
<?php
  function foo() {
  }
?>
```

Example 90: PHP Function

In CIL, the semantic equivalent is:

```
.method public static class object foo() {
  .maxstack 1
  ldnull
  ret
}
```

Example 91: CIL Method

Please note that each PHP function is translated to a public static method always returning a value of type **object**. In case a PHP function does not return a value, a **null** reference will be returned as can be seen in the example above; otherwise the value returned explicitly will be used.

Let us now extend the method with an argument.

```
<?php
  function foo($a) {}
?>
```

Example 92: PHP Function with Argument

Now the semantic equivalent in CIL is:

```
.method public static class object foo(object) {
  .maxstack 2
  ldarg 0
  ldstr "$a"
  call void class PHP.Runtime.Core::StoreToVariable
(object, object)
  ldnull
  ret
}
```

Example 93: CIL Function with Argument

For convenience sake all changes in the code are marked in italic. As one can see the argument **$a** now appears in the signature of the CIL method as **object**. Besides that a local variable called **$a** is created in the method body; the value passed to the argument when calling **foo** is loaded and saved to **$a**.

Optionally the argument has a default value assigned as follows:

```
<?php
  function foo($a = 2) {}
?>
```

Example 94: PHP Function with Default Argument

Now the semantic equivalent in CIL is:

```
.method public static object foo(object) {
  .maxstack 3
  ldarg 0
  dup
  brtrue passed
  pop
  idc.i4 2
  box System.Int32
passed:  ldstr "$a"
  call void class PHP.Runtime.Core::StoreToVariable
(object, object)
  ldnull
  ret
}
```

Example 95: CIL Function with Default Argument

Again details in the code that changed are marked as italic now. Now a check is inserted testing if the value passed as argument is a **null** reference. If so, the default value of the argument is used instead; if not, the default value is skipped and the one passed is used as before.

6.8 Classes and Objects

As mentioned earlier, handling of classes and objects is directly supported by CIL which makes translation of these constructs quite straight forward. Moreover, CIL even supports many more object orientated features than PHP in its current version with the result that not all of them will be needed during translation.

A typical class declaration in PHP looks as follows.

```
<?php
  class Circle extends Shape {
    public $radius;
    const PI = 3.14159265;
  }
?>
```

Example 96: PHP Class with Members

In CIL, the semantic equivalent is:

```
.class public Circle extends Shape {
  .field public object radius
  .field public static initonly object PI
  .method public void .ctor () {
    .maxstack 2
    ldarg.0
    ldnull
    stfld object Circle::radius
    ret
  }
  .method public static void .cctor () {
    .maxstack 2
    ldc.r8 3.14159
    box System.Double
    stsfld object Circle::PI
    ret
  }
}
```

Example 97: CIL Class with Fields

As can be seen, a default constructor **.ctor** will be added automatically if there is no user defined one. Such a default constructor is required for the creation of instances of the class. Members are translated directly as fields in CIL with the respective visibility declaration, of course. Constants are translated as fields being **public static initonly**. The existence of a static field is also the reason for the necessity of a default static constructor **.cctor** as both instance and static fields are initialized in the default constructors.

Methods of a class are translated directly as methods of the CIL class. Translation of functions is identical as described in chapter 6.7, apart from using the respective visibility declaration.

New instances of a class are created using the **newobj** instruction whereas parameters must be pushed on the stack beforehand and in the right order. Accessing an object's members is translated by accessing the respective fields.

So have a look on creating an object of type Circle and setting its radius afterwards:

```
<?php
  $c = new Circle();
  $c->radius = 2;
?>
```

Example 98: PHP Object Instantiation and Field Accession

In CIL, the semantic equivalent is:

```
1:  newobj instance void class Circle::.ctor()
2:  ldstr "$c"
3:  call void class PHP.Runtime.Core::StoreToVariable
(object object)
4:  ldstr "$c"
5:  call object class PHP.Runtime.Core::LoadFromVariable
(object)
6:  ldc.i4 2
7:  box System.Int32
8:  ldstr "$radius"
9:  call void class PHP.Runtime.Core::StoreToClassMember
(object, object, object)
```

Example 99: CIL Object Instantiation and Field Accession

First of all a new instance of class **Circle** is created in line 1. Lines 2 and 3 create a local variable named **$c** and store the instance just created to that variable. Further on, lines 4 and 5 load the contents of **$c** again, before in lines 6 and 7 the desired value **2** is loaded of type **System.Int32** and in line 8 an object of type **string** representing the name of the desired field **$radius** is loaded. The three objects on top of the stack are the arguments required for the runtime method **StoreToClassMember** which finally performs the storage operation desired in line 9.

The comparison of two objects for equality and identity is implemented using the runtime methods **IsEqual**, **IsNotEqual**, **IsIdentical** and **IsNotIdentical**, all of them taking two arguments of type **object**. Cloning is translated using a runtime method **Clone** taking the object to be cloned as argument of type **object** again. Since further elaboration of these methods would go beyond the scope of this thesis let me provide the indication that the complete source code and all detailed comments are available at [Rome2005].

7 mPHP – the Mono PHP Compiler

In this chapter I will present further details about the concrete compiler including information on its architecture, implementation, deployment, usage and an example. The compiler is called mPHP which is an alias for Mono PHP Compiler. This naming is derived from the Mono C# Compiler called mcs. Source code and binaries can be obtained at the SourceForge project page http://php4mono.sourceforge.net.

7.1 Architecture

The Mono PHP Compiler basically consists of two parts. There is a front-end translating the source language input into an intermediate representation whereas a back-end deals with the internal representation and produces code in the target language.

The front-end comprises two components. A scanner separates the PHP source code into tokens; afterwards a parser converts the input into an intermediate representation by building an abstract syntax tree (AST) of the tokens recognized by the scanner. Furthermore, it checks syntactical correctness, in particular if the code to be compiled is compliant with the PHP grammar. Semantic checks are also performed by the parser to the extent possible at compile time.

If the input successfully passed the front-end, the resulting AST will finally be passed to the back-end, which performs several optimizations on the AST, translates it to CIL and saves the result as assembly into an executable file. Any errors occurring during the process of compilation, are passed to an error handler and reported in a user friendly and comprehensible way.

An overview about mPHP's general architecture can be seen in the following diagram.

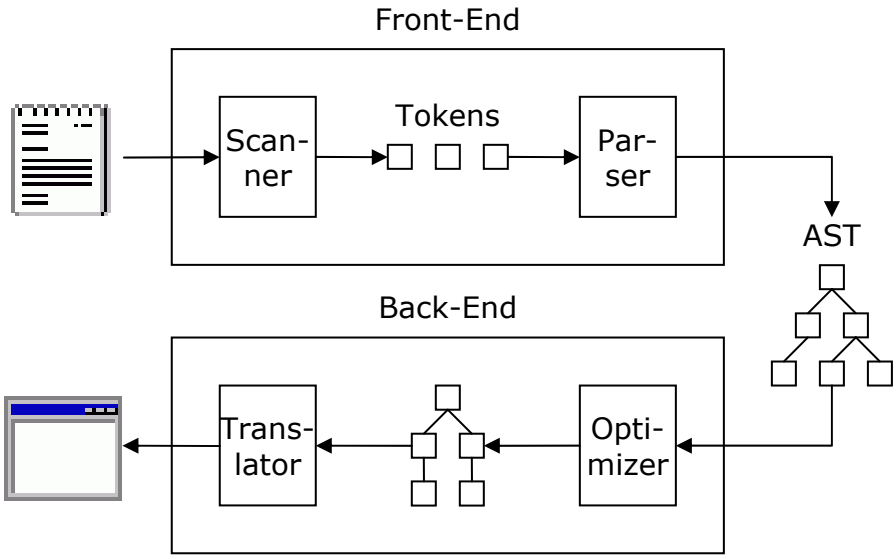


Figure 9: mPHP Architecture

7.2 Implementation

The whole compiler is written in C# as it is supposed to be compilable in Mono itself and C# is the language best supported by Mono up to now.

When implementing scanner and parser I decided to use generator tools for reasons of convenience and efficiency. Generators use structured and concise files specifying the syntax of the language to be processed, which can be modified easily. This guarantees the ability to modify scanner and parser for future versions of PHP in a most practical way. Furthermore, generator tools usually generate code that is more efficient than a basic hand coded scanner or parser would be. Another aspect is the way of parsing which can be done top-down or bottom-up. I decided for the latter one as this is the way the original PHP parser is implemented and major parts of it can be reused like that.

Of course the generator tools to be used need to produce C# code as this is the implementation language of the whole compiler. There are not many skillful tools available having that ability; in the end I decided for C#Flex [Csfl2004] as scanner generator and C#Cup [Imri2005] as parser generator. Both of them are open source C# ports of well-known Java generators.

In general there are two ways of processing the compilation. The first one is creating textual output containing all CIL instructions in a human readable format. This textual representation of the intermediate code must then be assembled to an executable file containing machine instructions which is done by an assembly linker. For this purpose Mono offers a special tool called **ilasm.exe**. The other way is the one I decided for; it means merging the two steps mentioned into one which is possible by using dynamic assemblies in Mono/.NET. The namespace **System.Reflection.Emit** offers such facilities. Like that modules, classes, methods and so on may be defined and created at compilation time and saved as permanent assembly.

7.3 Deployment

The Mono PHP Compiler consists of two parts. There is an executable called **mPHP.exe** containing all functionality to perform compilation and a library called **mPHPRuntime.dll** containing all runtime functionality needed to execute a compiled PHP script.

7.3.1 Executable

The executable **mPHP.exe** is deployed in three phases. The first phase consists of generating the scanner which is done by using the command

```
C#FLex --csharp --nobak mPHP.flex
```

Example 100: Generating Scanner

This will create a file called **Scanner.cs** from the scanner specification file **mPHP.flex** containing all functionality needed. Due to a bug in Mono, you have to modify the generated scanner file as Mono does not break the loop in line 1572 being **zzForAction: break;**. Just replace that break statement by an equivalent **goto** one.

In the second phase the parser needs to be generated which is done by using the command

```
C#Cup -parser Parser -symbols ParserSymbols -expect 1 mPHP.cup
```

This command will create a file called **Parser.cs** from the parser specification file **mPHP.cup** containing the parser's functionality. Furthermore, the numeric encodings for the parser symbols are enclosed in a newly created file **Parser-Symbols.cs**.

The third phase finally compiles the executable which is done by using the command

```
mcs /warn:1 /main:PHP.Compiler.MainClass /t:exe /out:mPHP.exe
<files>
```

In doing so, **<files>** refers to the generated files for scanner and parser and to all C# source files in the directories **src**, **src/PHP**, **src/PHP/Compiler** and **src/PHP/Compiler/CSCupRuntime**.

7.3.2 Runtime Library

The runtime library mPHPRuntime.dll is done using the command

```
mcs /warn:1 /t:library /out:mPHPRuntime.dll <files>
```

In this case **<files>** refers to the C# source files in the directories **src**, **src/PHP** and **src/PHP/Runtime** as these are the files containing all functionality needed at runtime.

7.4 Usage

The Mono PHP Compiler is distributed with a couple of files including the two essential ones **mPHP.exe** and **mPHPRuntime.dll**. There are other files included as well; however, they are only required for providing additional information and examples and therefore are not necessary to run the compiler.

The only requirement for running the Mono PHP Compiler is Mono in version 1.1.8.3 or higher. No installation procedure is necessary; just make sure the files **mPHP.exe** and **mPHPRuntime.dll** are located in the same directory. To ensure comfortable usage of the Mono PHP Compiler it is useful to ensure that your path contains the Mono binaries directory.

The command for compiling a PHP script looks as follows:

```
mono mPHP.exe [options] <source file>
```

Hereby **[options]** may be any of the following options:

```
-out:<file>             Specifies output file
-target:<kind>          Specifies the target (short: -t:)
                        <kind> is one of: exe, library
-reference:<file list>  References the specified assembly
                        (short: -r:) - files in <file list>
                        separated by , or ;
-nowarn                 Disables warnings
-help                   Displays this usage message (short: -?)
```

7.5 Example

In this chapter I will provide a short example showing how a compiled PHP script can be used from a class written in another .NET language.

In the following PHP script a class **MathPHP** including two functions is presented, one called **Fac** for calculating faculty numbers, the other one called **Fib** for Fibonacci numbers:

```
<?php
  class MathPHP {
    public static function Fib($a) {
      if ($a == 0)
        return 0;
      else if ($a == 1)
        return 1;
      else
        return self::Fib($a - 1) + self::Fib($a - 2);
    }
    public static function Fac($a) {
      if ($a == 0 || $a == 1)
        return 1;
      else
        return $a * self::Fac($a - 1);
    }
  }
?>
```

Example 101: MathPHP

Now we do not want to invoke them from inside the PHP script, but from a class written in another .NET language. Say the script given above is saved in

a file called **MathPHP.php**. Right now we will compile the class using the **/target:library** option:

```
mono mPHP.exe /t:library MathPHP.php
```

This will produce an assembly file called **MathPHP.dll**.

Now see a C# class below using the functionality of the compiled PHP script:

```
public class MathCS {
  public static void Main(string[] args) {
    // calculating faculty of 5
    object fac5 = MathPHP.Fac(5);
    System.Console.WriteLine("Fac(5) = " + fac5);
    // calculating Fibonacci of 5
    object fib5 = MathPHP.Fib(5);
    System.Console.WriteLine("Fib(5) = " + fib5);
  }
}
```

Example 102: MathCS

We will now compile this class using the Mono C# Compiler. Say it is saved in a file called **MathCS.cs**. As it uses features of the compiled PHP script, it's necessary to refer to the assembly we just created and to the mPHPRuntime:

```
mcs /r:MathPHP.dll,mPHPRuntime.dll /out:MathCS.exe MathCS.cs
```

This will produce the executable file **MathCS.exe**. Calling **mono MathCS.exe** will output the result as expected:

```
Fac(5) = 120
Fib(5) = 5
```

8 Prospects

At the end of this thesis I would like to dare an outlook into the future. How could it go on with PHP and Mono? An important point is the fact that this project does not cover compilation of all functionalities PHP offers. An overview on features covered is shown in Appendix B. Implementing the missing features is a desirable aim as it is the prerequisite for wider use of the Mono PHP Compiler. The feature probably most missing is access to external PHP libraries from a compiled script as these are widely used and a major reason for PHP's overall success. If these gaps in the current functionality are closed I am sure more and more people will join.

With Mono becoming available on more platforms, even on devices such as the brand new Nokia 770 more and more potential applications will turn up. Given Mono's ability to combine multiple languages, PHP code could be included into existing applications and run on platforms one would not have thought of before. As the most prominent open source implementation it is highly likely that the future success of Mono will be used to measure the open source and cross-platform success of .NET [EaKi2004]. Although it has been publicly debated whether Microsoft might attempt to strangle Mono with patent or copyright lawsuits, the very existence of Mono boosts the overall value of .NET and should flatter Microsoft by providing the CLI's versatility. Mono's roadmap has been heavily loaded with expectation. As the writer of this thesis I am personally convinced that Mono's success including the PHP Compiler is only a matter of time.

Appendix A CIL Instruction Set

This appendix specifies important instructions in Common Intermediate Language. For a complete listing please refer to [ECMA2005].

A.1 Arithmetic Instructions

Name	Description	Δ	Stack Transition Diagram
add	Adds **v1** and **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**
div	Divides **v1** by **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**
mul	Multiplies **v1** by **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**
neg	Negates **v** pushing the result on the stack	0	**..., v -> ..., res**
rem	Computes remainder of dividing **v1** by **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**
sub	Subtracts **v2** from **v1** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**

Table 16: CIL Arithmetic Instructions

A.2 Bitwise Instructions

Name	Description	Δ	Stack Transition Diagram
and	Computes the bitwise and of **v1** and **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**
not	Computes the bitwise complement of **v** pushing the result on the stack	0	**..., v -> ..., res**
or	Computes the bitwise or of **v1** and **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**
shl	Shifts **v** left by **n** number of bits pushing the result on the stack	-1	**..., v, n -> ..., res**

Name	Description	Δ	Stack Transition Diagram
shr	Shifts **v** right by **n** number of bits pushing the result on the stack	-1	**..., v, n -> ..., res**
xor	Computes the bitwise xor of **v1** and **v2** pushing the result on the stack	-1	**..., v1, v2 -> ..., res**

Table 17: CIL Bitwise Instructions

A.3 Control Flow Instructions

Name	Description	Δ	Stack Transition Diagram
beq <target>	Transfers control to **target** if **v1** is equal to **v2**	-2	**..., v1, v2 -> ...**
bge <target>	Transfers control to **target** if **v1** is greater than or equal to **v2**	-2	**..., v1, v2 -> ...**
bgt <target>	Transfers control to **target** if **v1** is greater than **v2**	-2	**..., v1, v2 -> ...**
ble <target>	Transfers control to **target** if **v1** is less than or equal to **v2**	-2	**..., v1, v2 -> ...**
blt <target>	Transfers control to **target** if **v1** is less than **v2**	-2	**..., v1, v2 -> ...**
bne <target>	Transfers control to **target** if **v1** is not equal to **v2**	-2	**..., v1, v2 -> ...**
br <target>	Transfers control unconditionally to **target**	0	**... -> ...**
brfalse <target>	Transfers control to **target** if **v** is zero (false)	-1	**..., v -> ...**
brtrue <target>	Transfers control to **target** if **v** is non-zero (true)	-1	**..., v -> ...**
call <method>	Calls the method **<method>** passing the required arguments from the stack and eventually pushing a return value on the stack		**..., arg1, ..., argN -> ...[, res]**
ret	Returns from current method eventually pushing a return value on the stack		**... -> ...[, res]**

Table 18: CIL Control Flow Instructions

A.4 Converting Instructions

Name	Description	Δ	Stack Transition Diagram
conv.i1	Converts **v** to **int8** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.i2	Converts **v** to **int16** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.i4	Converts **v** to **int32** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.i8	Converts **v** to **int64** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.i	Converts **v** to **native int** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.u1	Converts **v** to **unsigned int8** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.u2	Converts **v** to **unsigned int16** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.u4	Converts **v** to **unsigned int32** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.u8	Converts **v** to **unsigned int64** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.u	Converts **v** to **native unsigned int** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.r4	Converts **v** to **float32** leaving the result on the stack	0	..., **v** -> ..., **res**
conv.r8	Converts **v** to **float64** leaving the result on the stack	0	..., **v** -> ..., **res**

Table 19: CIL Converting Instructions

A.5 Load and Store Instructions

Name	Description	Δ	Stack Transition Diagram
ldarg <i>	Loads the **i**th argument of the current method on the stack	1	... -> ..., **res**
ldc.i4 <n>	Pushes the numeric constant **n** on the stack as **int32**	1	... -> ..., **res**

Name	Description	Δ	Stack Transition Diagram
ldc.i8 <n>	Pushes the numeric constant **n** on the stack as **int64**	1	**... -> ..., res**
ldc.r4 <n>	Pushes the numeric constant **n** on the stack as **float32**	1	**... -> ..., res**
ldc.r8 <n>	Pushes the numeric constant **n** on the stack as **float64**	1	**... -> ..., res**
ldc.i4.0	Pushes the numeric constant **0** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.1	Pushes the numeric constant **1** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.2	Pushes the numeric constant **2** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.3	Pushes the numeric constant **3** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.4	Pushes the numeric constant **4** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.5	Pushes the numeric constant **5** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.6	Pushes the numeric constant **6** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.7	Pushes the numeric constant **7** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.8	Pushes the numeric constant **8** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.9	Pushes the numeric constant **9** on the stack as **int32**	1	**... -> ..., res**
ldc.i4.m1	Pushes the numeric constant **-1** on the stack as **int32**	1	**... -> ..., res**
ldloc <i>	Loads the **i**th local variable of the current method on the stack	1	**... -> ..., res**
ldnull	Loads a null reference on the stack	1	**... -> ..., null**
starg <i>	Stores **v** to the **i**th argument of the current method	-1	**..., v -> ...**
stloc <i>	Stores **v** to the **i**th local variable of the current method	-1	**..., v -> ...**

Table 20: CIL Load and Store Instructions

A.6 Logical Instructions

Name	Description	Δ	Stack Transition Diagram
ceq	Compares **v1** and **v2** pushing **1** on the stack, if **v1** is equal to **v2** and pushing **0**, otherwise	-1	**..., v1, v2 -> ..., res**
cgt	Compares **v1** and **v2** pushing **1** on the stack, if **v1** is greater than **v2** and pushing **0**, otherwise	-1	**..., v1, v2 -> ..., res**
clt	Compares **v1** and **v2** pushing **1** on the stack, if **v1** is less than **v2** and pushing **0**, otherwise	-1	**..., v1, v2 -> ..., res**

Table 21: CIL Logical Instructions

A.7 Object Instructions

Name	Description	Δ	Stack Transition Diagram
callvirt <method>	Calls the method **<method>** associated with **obj** passing the required arguments from the stack and eventually pushing a return value on the stack		**..., obj, arg1, arg2, ..., argN -> ...[, res]**
castclass <class>	Casts **obj** to **<class>** pushing the cast object on the stack	0	**..., obj -> ..., res**
isinst <class>	Tests if **obj** is an instance of **<class>** pushing an instance of that class on the stack, if that is the case and pushing **null**, otherwise	0	**..., obj -> ..., res**
ldfld <field>	Loads the value of field **<field>** of object **obj** on the stack	0	**..., obj -> ..., res**
ldsfld <field>	Loads the value of static field **<field>** on the stack	1	**..., -> ..., res**
ldstr <string>	Pushes an object of type **System.String** for the string **<string>** on the stack	1	**..., -> ..., res**

Name	Description	Δ	Stack Transition Diagram
newobj <ctor>	Allocates an object of the type specified by the constructor **<ctor>** and calls **<ctor>** passing the required arguments from the stack	0	**..., arg1, ..., argN -> ..., res**
stfld <field>	Replaces the value of field **<field>** of object **obj** with value **v**	-2	**..., obj, v -> ...**
stsfld <field>	Replaces the value of static field **<field>** with value **v**	-1	**..., v -> ...**

Table 22: CIL Object Instructions

A.8 Array Instructions

Name	Description	Δ	Stack Transition Diagram
ldelem.i1	Loads the **i**th element with type **int8** of array **arr** on the stack as **int32**	-1	**..., arr, i -> ..., res**
ldelem.i2	Loads the **i**th element with type **int16** of array **arr** on the stack as **int32**	-1	**..., arr, i -> ..., res**
ldelem.i4	Loads the **i**th element with type **int32** of array **arr** on the stack as **int32**	-1	**..., arr, i -> ..., res**
ldelem.i8	Loads the **i**th element with type **int64** of array **arr** on the stack as **int64**	-1	**..., arr, i -> ..., res**
ldelem.i	Loads the **i**th element with type **native int** of array **arr** on the stack as **native int**	-1	**..., arr, i -> ..., res**
ldelem.u1	Loads the **i**th element with type **uint8** of array **arr** on the stack as **int32**	-1	**..., arr, i -> ..., res**
ldelem.u2	Loads the **i**th element with type **uint16** of array **arr** on the stack as **int32**	-1	**..., arr, i -> ..., res**
ldelem.u4	Loads the **i**th element with type **uint32** of array **arr** on the stack as **int32**	-1	**..., arr, i -> ..., res**
ldelem.u8	Loads the **i**th element with type **uint64** of array **arr** on the stack as **int64**	-1	**..., arr, I -> ..., res**
ldelem.r4	Loads the **i**th element with type **float32** of array **arr** on the stack	-1	**..., arr, i -> ..., res**

Name	Description	Δ	Stack Transition Diagram
ldelem.r8	Loads the **i**th element with type **float64** of array **arr** on the stack	-1	**..., arr, i -> ..., res**
ldelem. ref	Loads the **i**th element with type **object** of array **arr** on the stack	-1	**..., arr, i -> ..., res**
ldlen	Pushes the length of array **arr** on the stack	0	**..., arr -> ..., res**
newarr <type>	Creates a new array with elements of type **<type>** with length **n**	0	**..., n -> ..., res**
stelem.i1	Replaces element at index **i** of array **arr** with value **v** of type **int8**	-3	**..., arr, i, v -> ...**
stelem.i2	Replaces element at index **i** of array **arr** with value **v** of type **int16**	-3	**..., arr, i, v -> ...**
stelem.i4	Replaces element at index **i** of array **arr** with value **v** of type **int32**	-3	**..., arr, i, v -> ...**
stelem.i8	Replaces element at index **i** of array **arr** with value **v** of type **int64**	-3	**..., arr, i, v -> ...**
stelem.i	Replaces element at index **i** of array **arr** with value **v** of type **native int**	-3	**..., arr, i, v -> ...**
stelem.r4	Replaces element at index **i** of array **arr** with value **v** of type **float32**	-3	**..., arr, i, v -> ...**
stelem.r8	Replaces element at index **i** of array **arr** with value **v** of type **float64**	-3	**..., arr, i, v -> ...**
stelem.ref	Replaces element at index **i** of array **arr** with value **v** of type **object**	-3	**..., arr, i, v -> ...**

Table 23: CIL Array Instructions

A.9 Other Instructions

Name	Description	Δ	Stack Transition Diagram
dup	Duplicates top of stack **v**	1	..., **v** -> ..., **v**, **v**
nop	Does nothing	0	... -> ...
pop	Removes top of stack **v**	-1	..., **v** -> ...

Table 24: CIL Other Instructions

Appendix B PHP Features Included

PHP Feature	Included	Not Included
Types	boolean integer float/double string arrays object null mixed	resource number callback
casting	explicit implicit	
variables	local static global	variable variables predefined variables superglobals
constants	standard constants magic constants	
expressions	all expressions	except `…`
operators	all operators	except @
control structures	if else elseif while do for foreach switch try catch	declare include/require
functions	user defined functions	variable functions built-in functions external modules
classes and objects	inheritance overloading abstraction interfaces cloning	iterators reflection

Table 25: PHP Features Included

List of Abbreviations

ANDF ..... Architecture Neutral Distribution Form
API ..... Application Programming Interface
ASCII ..... American Standard Code for Information Interchange
ASP ..... Active Server Pages
AST ..... Abstract Syntax Tree
CIL ..... Common Intermediate Language
CLI ..... Common Language Infrastructure
CLR ..... Common Language Runtime
ETH ..... Eidgenössische Technische Hochschule
HTML ..... Hypertext Markup Language
HTTP ..... Hypertext Transfer Protocol
IEEE ..... Institute of Electrical and Electronics Engineers
IL ..... Intermediate Language
ISO ..... International Organization for Standardization
JIT ..... Just in Time
JVM ..... Java Virtual Machine
mcs ..... Mono C# Compiler
mPHP ..... Mono PHP Compiler
MSIL ..... Microsoft Intermediate Language
PHP ..... PHP: Hypertext Preprocessor
PHP/FI ..... Personal Home Page / Forms Interpreter
UCSD ..... University of California San Diego
VES ..... Virtual Execution System
XML ..... Extensible Markup Language

List of Figures

List of Tables

List of Definitions

List of Examples

Bibliography

[BenA2005] Jan Benda, Martin Maly, Tomas Matousek et al. Phalanger – the PHP Language Compiler for .NET Framework. 2005. http://www.php-compiler.net (accessed December 1, 2005).

[Csfl2004] N/A. C#Flex. 2004. http://sourceforge.net/projects/csflex (accessed December 9, 2005).

[DuBo2004] Edd Dumbill, Neil Bornstein. Mono: A Developer's Notebook. O'Reilly. 2004. Page 63 et seqq.

[EaKi2004] M.J. Easton, Jason King. Cross-Platform .NET Development: Using Mono, Portable.Net and Microsoft.NET. Apress. 2004. Page 461 et seqq.

[ECMA2005] ECMA International. Standard ECMA-335, Common Language Infrastructure (CLI). 3rd edition. 2005. http://www.ecma-international.org/publications/standards/Ecma-335.htm (accessed November 24, 2005).

[IEEE1985] Institute of Electrical and Electronics Engineers. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic. 1985.

[Girs2005] Ross Girshick. IronPHP. 2005. http://ironphp.sourceforge.net (accessed December 1, 2005).

[Goug2002] John Gough. Compiling for the .NET Common Language Runtime (CLR). Prentice Hall. 2002. Page 4 et seq.

[Imri2005] Samuel Imriska. C#Cup Manual. 2005. http://www.infosys.tuwien.ac.at/cuplex/cup.htm (accessed December 9, 2005).

[Know2004] Alan Knowles. PHP Sharp. 2004. http://www.akbkhome.com/wiki.php/Projects/PHP_Sharp (accessed December 1, 2005).

[Levi1995] John R. Levine, Tony Mason, Doug Brown. Lex & Yacc. 2nd edition. O'Reilly. 1995. Page 28 et seq.

[Merr2002] Brad Merrill. C# Lexer. 2002. http://www.cybercom.net/%7Ezbrad/DotNet/Lex (accessed January 16, 2006).

[MonA2005] The Mono Project. .NET Framework Architecture. 2005. http://www.mono-project.com/.NET_Framework_Architecture (accessed November 24, 2005).

[MonF2005] The Mono Project. FAQ: General. 2005. http://www.mono-project.com/FAQ:_General (accessed November 24, 2005).

[MonL2005] The Mono Project. Class Library Status. 2005. http://www.mono-project.com/class-status.html (accessed November 24, 2005).

[PhpU2005] The PHP Group. PHP Usage Stats. 2005. http://www.php.net/usage.php (accessed January 24, 2006).

[PhpH2005] The PHP Group. History of PHP and related projects. 2005. http://www.php.net/manual/en/history.php (accessed October 10, 2005).

[Rome2005] Raphael Romeikat. PHP4Mono – the Mono PHP Compiler. 2005. http://php4mono.sourceforge.net (accessed December 5, 2005).

[Trol1999] Trolltech AS, Norway. Q Public License v1.0. 1999. http://www.trolltech.com/licenses/qpl.html (accessed December 13, 2005).

[UCSD1995] University of California, San Diego. p-System: Description, Background, Utilities. 1995. http://www.ics.uci.edu/~archive/documentation/p-system/p-system.html (accessed November 29, 2005).

[WikC2005] Wikipedia. Class (computer science). 2005. http://en.wikipedia.org/wiki/Class_(computer_science) (accessed November 22, 2005).

[WikF2005] Wikipedia. Function (programming). 2005. http://en.wikipedia.org/wiki/Function_(programming) (accessed November 22, 2005).

[WikS2005] Wikipedia. Statement (programming). 2005. http://en.wikipedia.org/wiki/Statement_(programming) (accessed November 17, 2005).

[ZenM2005] Zend Technologies Ltd. PHP Manual. 2005. http://www.zend.com/manual (accessed November 7, 2005).

[ZenT2005] Zend Technologies Ltd. Technology. 2005. http://www.zend.com/zend/technology.php (accessed October 10, 2005).